Stellingen
behorende bij het proefschrift

Programming for Parallelism and Compiling for Efficiency

van Edwin Paalvast

1. Programmeertalen ondersteunen het creeren van een nieuwe (abstractere) virtuele machine slechts in het code-domein door het samenstellen van een verzameling functies. Ditzelfde zou in het data-domein mogelijk moeten zijn door middel van het definiëren van nieuwe afbeeldingen van datastructuren.

Hoofdstuk 4 van dit proefschrift

2. Wat de goto in het code-domein is, is de pointer in het data-domein.

3. Het programmeren met annotaties is alleen al vanwege het feit dat altijd correcte parallelle code wordt geproduceerd te prefereren boven expliciet parallel.

4. Het ontbreken van een orthogonalisatie van data organisatie and computation organisatie in Fortran-D beperkt de beschrijvingskracht.

Fox et al. Fortran-D language Specification

5. Neural computing heeft ons meer geleerd over onze cognitieve beperkingen dan de structuur van onze hersenen.

6. Het "markt-gericht" maken van (niet-)universitaire onderzoeksinstellingen leidt, indien dit niet gepaard gaat met een integrale aanpak, tot kwaliteitsvermindering en versnippering.

7. Bij de beoordeling van de geschiktheid van de promovendus voor het functioneren in de wetenschappelijke wereld zou naast het toetsen van het proefschrift ook een aanvraag voor onderzoeksgelden moeten worden beoordeeld.

8. Universiteiten gedragen zich meer en meer als beroepsopleiders door het invoeren van steeds nieuwe studierichtingen.

9. Dat decompositie niet altijd de beste methode is om tot efficiëntie verbetering te komen blijkt uit de economische toestand in de ex-Sovjet Unie.
Programming for Parallelism and Compiling for Efficiency

Proefschrift

ter verkrijging van de graad van
doctor aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. drs. P.A. Schenck
in het openbaar te verdedigen ten overstaan van een
commissie aangewezen door het College van Dekanen
op donderdag 11 juni 1992 te 10.00 uur

door

Edwin Martinus Rudolf Maria Paalvast

geboren te ‘s Gravenhage,
doctorandus in de Wiskunde en Informatica
Dit proefschrift is goedgekeurd door de promotoren:
prof. dr. ir. H.J. Sips
prof. dr. ir. L. Dekker

This research has been partly sponsored by TNO and SPIN.
Aan mijn ouders
en Brigitte
# Contents

## Preface

### Chapter 1 Introduction

- 1.1 The Demand for Processing Power ........................ 1
- 1.2 Parallel Machines .......................................... 2
- 1.3 Programming Parallel machines ............................ 4
- 1.4 The Research Question ...................................... 5
- 1.5 Dissertation Outline ........................................ 7

## Chapter 2 Aspects of Parallel Programming

- 2.1 From Problem to Program .................................... 9
- 2.2 Current Parallel Languages and Models .................. 11
- 2.3 Programming Environments ................................ 16
- 2.4 Research Question Revisited ................................ 16

## Chapter 3 Criteria for and Concepts of a Parallel Programming Language

- 3.1 Language Design Criteria .................................. 19
- 3.2 Language Concepts .......................................... 22
- 3.3 Influence of Language Concepts on the Design Criteria .. 24
- 3.4 Design Choices .............................................. 26
- 3.5 The Sequel .................................................. 29

## Chapter 4 The Booster Language

- 4.1 Overview of the Language ................................. 31
- 4.2 Shapes and Views ........................................... 33
- 4.3 View Operations ............................................ 38
- 4.4 Content Operations ......................................... 50
- 4.5 Control Statements ......................................... 62
- 4.6 Modules .................................................... 64

## Chapter 5 Exploiting the Parallel Virtual Machine

- 5.1 Relating Booster, Virtual Machines, and Real Machines 74
- 5.2 Annotation Language ........................................ 77
- 5.3 Data Organisation .......................................... 82
- 5.4 Computation Organisation .................................. 87
- 5.5 Exploiting Parallelism ..................................... 90
Preface

The work for this thesis was performed at the TNO Institute of Applied Computer Science (ITI-TNO) as part of the ParTool-project [Jong92], and has been partly subsidised by the SPIN funding agency. During this project I had many stimulating discussions especially with Arjan van Gemund of ITI-TNO and Leo Breebaart of Delft University of Technology. I am also grateful for the many discussions I had with other members of the ParTool project: Maarten van Steen, Teus Vogel, Peter Doornbosch, and Erwin Bakker of ITI-TNO, Edwin de Jong of University of Leiden, Henry Thomas a visiting scientist from IRISA in Rennes, France, Jan Verheul of Delft University of Technology, and Gerard Barkema of the University of Utrecht.

I am also grateful to my "promotors" Henk Sips and Len Dekker of Delft University of Technology who have inspired the research. Especially Henk Sips as my coach always encouraged me with everlasting optimism and new problems. Finally, I wish to thank TNO in general and Hans Overmeer in particular, for creating the environment in which this research took shape.

Some of the material of this thesis has appeared as publications. The following articles contain parts of Chapter 4 and Chapter 5.


The following articles contain parts of Chapters 6, 7, and 8.


The taxonomy of Chapter 9 is published in [Thomas92].


Edwin Paalvast
Chapter 1
Introduction

Computers are used to solve complex and computationally intensive problems. Because sequential computers fail to provide enough computation capacity, parallel computers are the only alternative to meet this demand. Parallel computers, however, by their very nature, have aggravated the already existing software-problem. These computers boost available computing-power, but they have made it more difficult to use that very power, because of the need to synchronise and to communicate results between co-operating processors. Hence, the programming of and compilation for this kind of computers introduces a whole new dimension.

Important for programming parallel machines is that the algorithm's intrinsic parallelism is being preserved when formalised in a programming language. However, most currently used programming languages do not have language constructs to express all potential parallelism. Consequently programs written in those languages contain sequential parts not induced by algorithmic constraints, but due to programming languages' shortcomings.

Along with the problem of effectively programming a parallel machine, the use of these machines also complicates the portability-issue or parability. Parability aggravates the already existing portability issue on sequential machines, because of the many degrees of freedom possible in architectures for parallel machines. Among these degrees of freedom are the number of processors, the processor-memory pairing, and the interconnection network. Nevertheless, we have to solve the parability-problem before parallel machines can become a practical alternative for sequential machines.

In this thesis, we define a programming language, which allows for the description of portable parallel programs in the field of scientific computation. We will research the translation of this language, with the goal to produce efficiently executing programs on a variety of parallel machines. Furthermore, we will show that the method used to formalise the semantics of the language and its translation is also applicable to comparable programming languages.

11 The Demand for Processing Power

There is an insatiable need for processing power in various fields of science and engineering. Today's computers are a thousand times faster than computers a decade ago, but we still have problems that take days, weeks, or even years to compute. However, in spite of the demand in increased speed, budgets tend to
remain at a constant level. Therefore, the cost of hardware and the relative cost of producing or adapting software for a new generation of computers are of paramount importance to the success of a particular system.

Various performance constraints motivate the need for more processing power. For example, applications for weather forecasting, fluid dynamics, and modelling of chemical reactions need fast computers to produce accurate data within reasonable time. In addition, for a number of applications this data must be available within a strict time-bound period to be of any use at all. True examples of this are weather forecasting and earthquake prediction. In other systems, such as computer aided design (CAD) or computer integrated manufacturing (CIM) systems, the need for faster computers has an economic motivation. To evaluate a number of product designs in a short time is essential for the ability to quickly adapt products to ever changing customer demands.

The applications mentioned above typically involve computations on large sets of data. We distinguish two types of computations: those that are compute-bound, computations that require much more time for computation than for data transport, and those that are I/O-bound, computations that require much more time for data transport than for the actual computation. Our focus will mainly be on the compute-bound applications.

1.2 Parallel Machines

Sequential — von Neumann type of — computers cannot deliver the required processing power for such applications. There speed and size is limited by the speed of light and maybe more important, engineering constraints. Although many improvements and innovations are still possible, the improvement in speed will not be sufficient to meet the demand.

Parallel machines, in principle, do not have an upper limit to the increase in speed; one can just add processors. Of course, actual parallel machines have built-in limitations as well, such as the bandwidth and speed of certain parts of the system, but the potential surpasses that of sequential machines.

In our discussion on programming and compiling for parallel machines, we need a categorization or taxonomy for parallel machines to distinguish between the various machine designs. Flynn [Flynn66, Flynn72] presented a first and still frequently used taxonomy for parallel hardware, which uses the pairing of instructions and data for classification. This taxonomy knows four categories: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), and MIMD (Multiple Instruction Multiple Data). An example of the first category is the sequential machine, array processors are an example of the second category, and an example of the fourth category is a machine interconnecting "independent" processing elements.
A disadvantage of this taxonomy is, amongst others, its inability to distinguish between memory hierarchies in parallel machines. More elaborate taxonomies, such as [Dasgupta90] and [Skllicorn90] also take memories and the interconnection topology of machines into account. However, these taxonomies only distinguish between parallel hardware, whereas for a more elaborate description of systems we need to take the various software layers, such as operating systems and I/O subsystems into account. In Chapter 5, we will return to this issue, but first we will pay attention to an important distinguishing characteristic for MIMD machines.

In MIMD computers each processor is capable of performing its tasks independently from other processors. In this class, we distinguish two types of machines: those with shared-memory and those with distributed-memory. In the first case, access to arbitrary data elements takes unit time, whereas in the latter case access to a local memory is faster than to a distant memory. Figure 1 illustrates both types of machines.

![Diagram](image)

*Figure 1. Shared- and distributed memory parallel machines*

Shared-memory machines have the drawback of having a poor scalability. A shared-memory parallel machine with thousands of processors is hardly feasible due to the problems that the fully connected network poses. The technological problem is that it requires the number of interconnections to be the square of the number of processors. Moreover, the growing cost of the required (very fast) interconnection network affects the price/performance ratio of these machines. Therefore, in the long term a parallel machine with some kind of memory distribution is the only alternative to achieve a substantial increase in processing speed, because here one can use the economy efficiency of access differentiation.

*Programming Interface to Parallel Machines*

Most shared-memory parallel machines have a task-oriented or multi-threaded programming model. The program, a collection of tasks, is mapped on a set of processors and the data (and programs) reside in the single shared memory accessible to all processors. A distributed-memory machine usually
has a message-passing programming model. Programs on each processor can access their local memory directly, but have to exchange messages to access memories not local to the processor.

Another programming model for a distributed-memory machine is the PRAM [Schwartz80] or virtual shared-memory model [Li89] which consists of a software-layer that hides the distribution of memories. The advantage of this approach is that it presents the programmer a unified programming model. The drawback is that for a large class of program-machine combinations the approach is too slow in comparison to, for example, explicit message passing.

Trends in machine architectures

In the last decades we have seen the design and development of a large number of different types of architectures, but no single type of architecture has emerged as the most favourable one, nor is it clear which architecture eventually will. We can, however, distinguish some trends and common characteristics that are of importance:

• More and more off-the-shelf components are used as basic processing elements;

• Parallel machines must be scalable, both regarding the number of processors and the total system bandwidth. Scalability of processors suggests some form of memory-distribution because the production of shared-memory systems with thousands of processors is hardly feasible. Scalability of the total system bandwidth suggests some form of memory hierarchy in order to transport and store the data at the required speed;

• There is a trend towards delay insensitive communication systems. In these systems, the distance between sender and receiver does not dominate the transport-time of a message. This property results in transparent system behaviour regarding the interconnection network.

1.3 Programming Parallel machines

Until now, shared-memory parallel machines have been accepted on a much wider scale than distributed-memory parallel machines. Since shared-memory machines, like sequential machines, have a single address space, migration of software from sequential to shared-memory parallel machines is less cumbersome than to distributed-memory machines. The latter are more difficult to program, because they require the programmer to use the message-passing paradigm. Programming in this paradigm is complex and prone to error, because the programmer must use the available resources efficiently by decomposing the program accordingly.

As a consequence, when programming distributed-memory parallel
machines, one must be aware of and make use of the locality of data and computation. Support for the programmer is imperative, because of the complex nature of this type of programming. We will discuss contemporary programming of parallel machines and its problems in Chapter 2.

Conclusions and premises

Although a decennium ago parallel programming was thought to have a very bright future, it has yet to fulfil its promise. Reasons for this are the wide variety of different types of parallel machines, the absence of standard programming models, and the amount of effort needed to optimise a parallel program for a certain machine, due to the lack of supporting tools and methods. Below, we summarise the current state of programming parallel machines:

- Parallel programming is often machine specific: both parallel language extensions and tools used to support the tuning of programs focus on specific machines. Porting programs to a different type of machine often involves the rethinking and reprogramming of large parts of the program. Most notably, this prevents easy experimentation with alternate algorithm designs;

- Parallel programming forces the programmer to have a fairly detailed knowledge of the underlying machine, its operating system, and the implications of certain (language) constructs on performance;

- Debugging of parallel programs is hard, especially on distributed-memory machines;

- In most application areas, there is an enormous amount of existing (sequential) software. Migration of (part of) this software to parallel machines would imply an unprecedented effort. A major conversion to these machines is only economically feasible, when the application runs significantly faster and the parallelised version can outlive the economic life-span of the application.

1.4 The Research Question

Before we pose the research question underlying this thesis, we will investigate the users' requirements for a programming environment or programming language and underlying parallel machine.

Requirements

The main interest of users of large computers is the solution of their problem and not the computer-intrinsics. Hence, users expect support on their level of thinking, for example on the high abstraction level of partial differential equations, flow equations, etc. The user would rather remain on that level of
problem statement. Current practice, however, learns that this goal is not yet achievable and that users have to pass the levels from the algorithm-level down to the program level. We will use the notion ease of programming when referring to the raising of the level of abstraction in programming.

A second user-requirement is the full exploitation of capabilities of computers. Especially in the field of natural science, the demand for more computer-power and data storage is insatiable. A problem of a certain size that computes within reasonable time on a state-of-the-art computer, may require much more time because of a simple refinement or extension to the problem. It is therefore essential not to waste any computer-power by poor use of resources. We refer to this requirement as efficiency.

Finally, the institutes or companies that use powerful computers usually have a wide range of different types of architectures. In addition, users have to migrate their programs frequently because of the rapid developments in the field of parallel machines. This implies that these programs must be portable between different types of parallel machines. The parability issue has an extra dimension when compared to the porting of programs between sequential machines. The issue of preservation of efficiency is much more prominent than for the sequential case.

In summary, from the user's perspective a programming model has to meet the three possible conflicting goals of ease of programming, efficiency, and parability. For an even better understanding of this perspective, we need to distinguish between various groups of users. Although the ultimate goal is a fully automated transformation of a user's problem to a (parallel) program, we are still in the phase of understanding parallel programming. Users will not only include mathematicians or algorithm designers, but also computer scientists and hardware designers. They also need tools and formalisms to program parallel machines, but they need to have more influence on the translation in order experiment with new ideas and algorithms.
From this, we formulate the research question of this thesis as follows:

*Does there exist a programming language that satisfies the demands of several types of users, that allows for the description of algorithms for scientific applications and ports to different types of parallel machines with relative ease, yielding efficient executing programs?*
1.5 Dissertation Outline

The title of this thesis "Programming for Parallelism and Compiling for Efficiency" suggests a subdivision of the work into programming on the one hand and translation aspects on the other. This subdivision is reflected in Chapters 3, 4, and 5, which deal with the programming aspects of parallelism and Chapters 6, 7, and 8, which discuss semantics, translation, and optimisation of programs. Chapter 2 discusses some aspects of parallel processing that are essential for the understanding of sequel of the thesis. Chapter 9 concludes with a summary of results and future directions.

Figure 2 presents the precedence graph for a successful reading of this thesis.

![Figure 2. The precedence graph for reading this Thesis](image-url)
Chapter 2
Aspects of Parallel Programming

In this chapter we continue the discussion on parallel programming. This problem is not only complex in the literal sense, but also has a wide variety of programming approaches, machines, and corresponding terminology. Before we give a more detailed formulation of the research question, we establish a frame of reference for programming parallel machines. We use this frame of reference in combination with the three criteria posed in Section 1.4 to discuss current (parallel) programming languages and models. We conclude the chapter with a renewed look at the research question.

2.1 From Problem to Program

We identify four levels of abstraction when going from problem statement to problem-solution (adapted from [Jamieson87]). At the first level, we have the identification of the problem: the problem statement, for example, a Finite Element problem in mechanical engineering. A number of approaches exist to solve these kinds of problems and they are formulated at the second level: the algorithm approach. For example, the solution to the problem can be specified in terms of the solution of a set of linear equations, but various ordering methods and various algorithm approaches can be used. Having identified a certain approach to solve the problem, the next and third level involves the description of an ideal algorithm. An ideal algorithm describes the computational steps to be performed, but a mapping to a specific machine has not yet been defined. In our example, depending on the ordering method selected in the algorithm approach, a certain algorithm is chosen to solve the set of linear equations. The next step is conceptually split in two: the definition of a virtual program for a specific unconstrained type of architecture and a (machine-dependent) program; an architecture-dependent realisation of an ideal algorithm in a programming language. A virtual program for a certain type of architecture and a (machine-dependent) program are related, but in the latter much of the parallelism may have been lost due to a choice of programming model and architectural constraints. In other words, the top-down approach from problem statement to a realisation of the algorithm in a program can be "disturbed" by discrepancies between the architecture of algorithm, the real machine, and deficiencies in the description formalisms on the various levels. The process from problem statement to machine-level has been depicted in Figure 3.
Figure 3 illustrates the problem-implementation cycle, where the process of gradual refinement leads to a better exploitation of the target parallel machine. When a given problem instance does not contain enough parallelism, one can try to follow a different algorithm approach, find better algorithms for a certain approach, or change the program, that is: the description of the algorithm.

The phases problem statement and algorithm approach are still in the informal paper and pencil stadium. Mathematics can be used to describe a certain problem instance, but this is rarely used to produce executable solutions in practice. Common practice today, however, involves manual translation from problem statement down to an algorithm implementation in a programming language tailored towards a specific parallel machine. The major problem encountered when describing an algorithm in a specific language is assessing the consequences of chosen combinations of language constructs on the overall performance of a particular program. In general, the detection of performance bottlenecks of a certain program on a large parallel machine exceeds the mental capabilities of human beings. As a consequence, the programmer needs some form of automated support.

We will give an overview of current approaches to programming parallel machines and programming environments aimed to support parallel programming.
2.2 Current Parallel Languages and Models

The most distinguishing characteristic of a paradigm for a parallel programming language is the way in which parallelism is expressed. Parallelism can either be expressed *explicitly* in a programming language or can be derived *implicitly* from a language by a software system (compiler, run-time system, or operating system). The latter category can again be subdivided in those approaches that do not have any notion of an underlying machine and those that allow a set of *pragma's* or annotations to be added to control and express parallelism. This last category is referred to as *enhanced implicit* programming languages. These languages are discussed in more detail below, commenting on how far they conform to the criteria: *efficiency, portability, and ease of programming.*

*Explicit Parallel Programming Languages and Models*

We find it important to make a distinction between parallel and concurrent programming languages. When a language is tailored to the description of several independent and distinct tasks that co-operate concurrently, we will refer to this language as a *concurrent* language. When a language is more suited for describing a single task with multiple executable instances in parallel, we will refer to this language as a *parallel* language.

Below, we will discuss various explicit programming languages and models in more detail. The discussion is not an attempt to give an exhaustive overview of all contemporary approaches, but merely presents representatives from different categories of parallel languages. For a more exhaustive survey of concurrent and parallel languages the reader is referred to [Bal89a].

*Languages with Extensions*

The introduction of parallel machines has been followed by the extension with parallel constructs of already existing programming languages for sequential machines, such as fork-join, doall, semaphores, monitors, and/or various message-passing constructs.

Most vendors of parallel machines ship their hardware not only with operating systems, but also with specific parallel language extensions to, for example, Fortran, C, and Lisp. For an overview of various extensions to Fortran, the reader is referred to [Karp88]. These language extensions have the disadvantage that porting of a program to a different machine at least involves the replacement of all machine-specific language extensions. Therefore *parability* is non-existent for these types of languages. Furthermore, these extended languages are not always well suited to express various forms of parallelism,
because of their imperative nature, which seriously hampers ease of programming.

Efficiency depends on the quality of the compiler and the skill of the programmer to circumvent the previously mentioned imperative nature of these extended programming languages. Because these languages are made up of two separately engineered language domains: the description of computation on the one hand and communication and synchronisation on the other, the optimisation process for a compiler is very difficult.

Subroutine or Macro Extensions to Languages

To overcome the problem of machine-specific parallel language extensions, a number of attempts have been undertaken to produce language and machine independent extensions. The Force [Jordan89] is an example of a set of machine-independent extensions to Fortran for shared-memory type of machines. For distributed-memory machines extensions such as Express, CrossOS [Fox88], and GMD/Argonne Macro's [Bomans90] have been developed. These extensions help to overcome part of the parability problem. It should be noted that most extensions are usually only available on or optimised for a limited number of machines and programming languages. In addition, because extensions are "hidden" for the compiler they cannot be subject to any optimisation or transformation procedure and consequently do not solve the efficiency problem.

Parallel Languages

The introduction of parallel machines also induced the design of new parallel and concurrent languages which incorporate one or more of the aforementioned programming constructs within the language. Most of these new languages are focused on concurrent programming, such as ADA [ADA83], Concurrent Modula [Mühl88], Concurrent C [Gehani86], C* [Quinn89], and Concurrent PASCAL [Brinch75]. In addition, a language like POOL-T [America87] offers concurrent programming with an object-oriented paradigm.

Other languages have been designed for more massive parallel programming and have been inspired by the type of parallel machine on which they are executed. The OCCAM language [Jones88] has been designed on the basis of CSP [Hoare85] and is used for programming Transputer systems. The ACTUS language [Perrott87] has been designed for the DAP and array-processors in general. PCF-Fortran [PCF90] is meant as a new standard for programming "Fortran" on shared memory parallel systems.

Although new language designs present a more consistent programming model to the programmer, each language is only implemented on a limited
number of parallel machines. Much as in sequential programming, there does not exist a single commonly accepted parallel programming language. Hence, the parability issue is not solved.

Compared to the language-extensions discussed above, parallel languages are more easy to translate and to optimise. A disadvantage is, however, that most languages are either aimed towards a task-oriented programming model (for shared-memory machines) or to a message-passing programming model (for distributed-memory machines). This therefore violates the parability criterion.

Superimposed Parallel Models

Another approach is to enhance an existing programming language with a superimposed explicit parallel programming model. A restricted number of parallel primitives and a shared data-space notion are added to a language such that the expression of parallel algorithms is more convenient. A number of approaches have taken the existence of multiple (sequential) programming languages as a starting point and have enriched these languages with a parallel programming model. Examples of this approach are Linda [Carriero89a+b], Orca [Bal89b], and The Paralation Model [Sabot88]. These superimposed parallel models involve the definition of a new (shared) data structure and associated access mechanism. In the case of Linda the new data structure is the tuple space and the access mechanism is pattern matching. For The Paralation Model, the paralation is the superimposed data structure and match and move primitives are used to access and manipulate paralations. The elwise construct is used for expressing parallelism.

Linda and the Paralation model have the advantage of adding, in a restricted way, parallelism to a familiar programming language, thus facilitating acceptance. A disadvantage is that the base languages are originally designed for sequential machines and may cause programmers to think sequential. Furthermore, the programmer must choose in which data-space the data is to be stored: that of the base language or that of the superimposed parallel model. Exchanges between the two require a new set-up of the program with respect to parallelism.

Implicit Parallel Programming Languages and Models

With the introduction of parallel architectures it was perceived that some languages by their nature could circumvent the problems of explicit parallel programming. Parallelism was already — implicitly — enclosed in the programs described in these languages. Pure functional languages are a good example of an implicitly parallel language. This insight of implicitly enclosing parallelism has sparked the design of languages such as data flow or single-
assignment languages: procedural style languages with certain properties. Another category of languages that are (partially) suited for the extraction of parallelism are those that are centred around a single data structure.

Functional Languages

In his well-known paper [Back78], Backus reasoned that most languages are based on Von Neumann type of machines, which hampers the programming of machines with an entirely different architecture. To this purpose, he introduced the language $FP$, which is based on a mathematical framework.

The same language properties: freedom of side-effects and the Church-Rosser property (no fixed evaluation order) are shared by other already existing languages such as LiSP [McCarthy60]. With the growing popularity of parallel architectures, functional languages have been developed that can be used to describe machine-independent (parallel) programs. Examples are Crystal [Chen85, Chen88], ParAlf [Hudak88, Hudak89], Miranda [Turner85] and Haskell [Hudak89].

The translation of functional programs with the goal of obtaining efficient running programs, however, remains a problem. Although much effort is put into the optimisation process, still compared to languages with explicit parallelism the results are poor. More on this subject can be found in [Jones87] for functional languages in general and in [Ryder88] for the translation of FP.

Dataflow or single assignment languages

A dataflow or single assignment language has, as a result of the interplay between syntax and semantics, the following properties: no side-effects, locality of effect, parallelism is generated by unfolding iterative computations, and no state information is retained between different invocations of functions and procedures. Examples of these languages are Val [McGraw82], Sisal [Sked85] [Lee88], and Id Nouveau [Arvind88].

A drawback of data-flow languages is that they assume specialised hardware. As the trend in parallel hardware is directed towards off-the-shelf products, the introduction of special hardware is a problem. In addition, dataflow machines do not scale very well [Hwang84].

Single assignment languages have been developed for shared-memory parallel machines. A problem for single assignment languages is that efficiency compared to explicit programming languages has been poor. Programs written in these languages tend to waste memory space due to the single assignment rule (e.g. the excessive copying of data arrays), although some work has been done to eliminate this problem [Gopinath87] [Ghara88].
2.2 CURRENT PARALLEL LANGUAGES AND MODELS

Data structure oriented languages

A number of languages has been developed that are centred around a single data structure. Although these languages have not been developed for parallel machines, some of them are declarative by nature and allow for extraction of implicit parallelism. Amongst these are languages which have the generalised array as basic central data structure, such as APL [Iverson62], FAC [Perlis86], and QNIAL [Jenkins86]. Especially for APL attempts have been made to compile this language for various vector and parallel machines [Koster85, Budd88a, Budd88b].

Enhanced Implicit Programming Languages and Models

Supported by automatic or interactive program restructurers, attempts have been undertaken to parallelize existing programs written in imperative languages. Automatic restructuring of (original) sequential programs for distributed-memory parallel machines similar to that of shared-memory systems has proven to be very difficult or only applicable to a limited set of programs.

In parallel with the research described in this thesis an approach, starting with [Callahan88a, Kennedy89], has been followed to aid the compiler by adding information on decomposition of the program and sometimes data. Concurrent with the above approach Zima and Gerndt in the Fortran language extensions for SUPERB [Bast89, Gerndt89a, Gerndt89b]. Most of these approaches are based on existing languages such as Fortran and C, which are extended with syntactic constructs. A limited number of data decompositions, such as block and block-overlap, can be defined on arrays and with help of the programmer be transformed to a parallel program in SUPRENUM Fortran. A slightly different approach has been followed for C by André and Thomas in their system Pandore [André90] [Thomas91]. They introduce a virtual machine concept to which the data decompositions are related. A similar approach but then applied to Id Nouveau is taken by Pingali and Rogers [Rogers89]. In [Rühl90] the Oxygen system is described that especially focusses on run-time aspects of annotating programming languages.

Kali-Fortran [Koelbel90] is an extension to Fortran which includes a forall statement and syntactic constructs to define data decomposition. A similar approach is found in Dino [Rosing88] that has language extensions to define the locality of data. In the approach taken in Fortran-D [Fox90] by Fox, Kennedy, et al., the programmer specifies the distribution of the data by aligning each array to a virtual array, which is then in turn distributed on the real machine. Vienna-Fortran [Chapman91] is also an extension to Fortran in which a direct mapping from a data structure to the real machine can be defined.
Most of the approaches mentioned above are based on Fortran or C. These imperative languages are not suited to express all the required parallelism and therefore for some types of problems efficiency or parability may cause a problem. Furthermore, much of the above approaches are still in the research phase and not many commercial compilers have emerged.

We will return to each of these approaches in more detail in Chapter 9, after we have established a more formal framework. This framework will be used to compare languages and to evaluate their adherence to the criteria introduced in the research question in Section 1.4.

2.3 Programming Environments

An appropriate language for the expression of parallelism is a first requirement for exploiting the possibilities of a parallel system. The sheer complexity of parallel programming, however, makes it impossible for programmers to predict what the effects of certain program fragments will be on overall performance. It is therefore necessary to have an integrated environment with tools to support the development of parallel programs allowing the performance assessment of programs.

The programming environments that have been developed in the last decade mainly focus on the parallelization and development of Fortran programs for shared-memory architectures. Examples of such environments are ParaScope [Callahan88b], Parafrase-2 [Polychr89], PTOOL [Allen85, Allen86], PTRAN [Allen88a, Allen88b], Start/Pat [Appelbe89], FAUST [Guarna89] and PISCES (2) [Pratt85, Pratt87].

Programming environments for distributed-memory architectures are the SUPERB [Zima88] system which takes annotations as input and allows interactive parallelization of the program. The last type of programming environments is still in the research stadium and much effort has to be invested in the optimisation process.

2.4 Research Question Revisited

In this chapter, we have established a frame of reference for parallel programming and have given a survey of contemporary languages and programming environments. Along with the survey, a number of deficiencies have been identified in various programming languages and models with respect to parability, efficiency, and ease of programming.

The conclusion is that in general explicit parallel programming may yield efficient running programs, but they require detailed knowledge of the parallel machine which hampers ease of programming. Furthermore, problems are encountered when porting programs written in these languages, because of the explicit parallelism.
Implicit parallel programming on the opposite results in portable, easy to write programs, but in general does not yield efficiently running programs. The third approach, enhanced implicit parallel programming, tries to enhance the translation by supplying the compiler with information on the decomposition of the program and data. This approach combines properties of the previously mentioned approaches. Programs are easily ported, but may require some effort in changes to the decomposition of the program. They result in reasonable efficient running programs and compared to the explicit parallel programming model are fairly easy to program. Unfortunately these languages and approaches still incorporate the drawbacks induced by their base languages.

In order to address the research question posed in the previous chapter properly, we need to have a closer and more detailed look at language concepts that make up a (parallel) programming language and the effect of these concepts on the criteria parability, efficiency, and ease of programming. We therefore add two, more detailed questions:

What language concepts or combinations of language concepts are most favourable to the criteria parability, efficiency, and ease of programming?

Can we define a language on basis of these concepts?
Chapter 3
Criteria for and Concepts of a Parallel Programming Language

We now postulate the language concepts for a parallel language satisfying the parability, efficiency, and ease of programming criteria as posed in the research question. To get a well-balanced language, we will formulate a number of additional and more detailed criteria concerning software engineering aspects of the language. These criteria serve to select the most favourable language concepts.

3.1 Language Design Criteria

Although the design of a programming language very much remains an art, a language must satisfy a number of underlying criteria. Here, we distinguish three kinds of design criteria: those that are valid for arbitrary (parallel) programming languages, those specific for a model of computation that consists of Von Neumann type of basic engines, and criteria related to the application domain. After formulation of these three types of criteria, we evaluate the various categories of languages mentioned in the previous chapter, to assess whether they comply with these criteria.

3.1.1 General Design Criteria

[Ghezzi87] contains a number of criteria for (sequential) programming languages. Successful parallel programming and translation, however, require more specific criteria. Apart from the already given criteria Parability, Efficiency, and Ease of Programming, we distinguish the following criteria:

*Information Hiding*

One of the achievements of the last decades in language design is the insight in the importance of information hiding in software development and maintenance.

*Abstraction*

To capture recurring patterns and to avoid low-level details we use abstraction. Raising the level of abstraction can also ease the programming burden. It is often advocated [Back78], [Chen85], [Hudak88], [Jones87], that a
declarative programming style is better suited to express the inherent parallel properties of algorithms than an imperative programming style.

There is, however, one kind of danger involved with abstraction: if one does not allow for the description of certain information known to the user, a compiler cannot perform certain optimisations. We therefore formulate the following criterion:

*Adding and Preservation of Information*

The language should allow for the (optional) description of information needed by the compiler to produce efficient translations. The nature and amount of additional information depends largely on the current state-of-art in compiling techniques. However, some users want to get the most out of a (parallel) machine and are willing to supply as much information as needed to achieve that goal.

Although some of the parallel languages discussed in the previous chapter allow for the specification of additional information, e.g. explicit communication, this information is intermingled in the program. Consequently, the program is not portable and it is not possible to add information incrementally. For example, one must describe a communication framework for a program as a whole and not incrementally. Furthermore, the cost in terms of time needed to program this additional information is not proportional to the gain, thus:

*Balance of Gain in Performance and Cost*

Parallel programming practice today involves much experimentation with different realisations (programs) of the same algorithm, before reaching a satisfactory solution. Simple experimentation with various data- and computation decompositions is therefore important. A major performance improvement justifies some programmer's effort, but it should be balanced.

Debugging of explicitly parallel programs with communication and synchronisation on a parallel machine is difficult, because the program contains computations interspersed with parallel primitives. We therefore pose the following design criterion:

*Separation of Concerns or Orthogonality*

The problem of parallel programming is difficult regarding program termination (deadlock, fairness), and program correctness issues. Therefore, a programming language needs to separate these issues.
3.1 LANGUAGE DESIGN CRITERIA

The following design criteria, adapted from [Mac83], are of a more general nature.

*Simplicity*
A language or model should be as small and simple as possible, with a minimal number of concepts and rules to combine these concepts.

*Zero-One-Infinity*
The only reasonable occurrences of elements in a language are zero, one, or an infinite number of times. For example, the dimension of array declarations may not be bounded, nor may operations on arrays be restricted to a certain (number of) dimensions.

*Definiteness*
The meaning and effect of programs in a programming language must be defined unambiguously. This includes the influence of the use of certain language constructs on program efficiency. Consequently, we need a formal definition of the semantics as well as the translation process.

### 3.1.2 Criteria Related to the Adapted Model of Computation

Since we assume that the basic computing engine in a parallel machine will be a Von Neumann type of machine, we can formulate two additional criteria that are especially relevant for parallel language design. The first criterion raises the *scalability* issue both with respect to the number of processors as well as the size of the input. The second criterion anticipates the fact that processing elements may contain *vector* processing capabilities.

*Scalability*
The scalability issue comprises of variable granularity and parameterized decomposition of computation and data. Variable granularity is necessary, because fixing the granularity in problem decompositions can lead to very bad performance figures. An efficient decomposition is dependent on the characteristics of certain elements of the parallel machine. Hence, only programming languages that allow the granularity to be varied can meet these requirements.

Since it is not always known what the size of data structures is at compile-time, a parallel language should allow for any parameterized decomposition, both to the dimension of the problem as well as to the number of processors involved.
Preservation of coherence information
In many algorithms, data structures are traversed in some linear fashion. This can often be exploited to generate very efficient code on vector as well as scalar machines. The available coherence information in the computation should therefore be preserved for efficient code-generation. This criterion is an extension to the previously formulated criterion on preservation of information.

3.1.3 Criteria Related to the Application Domain
In the application domain of scientific computing, data is usually structured by some kind of grid, which may represent a real life object such as a plane or a building. Arrays are often used to represent these grids. The index of the array identifies the grid number and the content of the array the associated value. Depending on the application these arrays may vary in size, structure, and sparsity. We therefore formulate the following criterion:

Data structures to describe grids
The language should incorporate array-type of data structures, in other words an absolute addressable data structure.

Algorithms that use these arrays often require complex management of subparts of the array and therefore constructs to describe and manipulate these subparts are necessary:

Constructs to manipulate arrays
Constructs to describe subparts of arrays, or a union of a set of subparts, should be available. Furthermore, it should be possible to easily manipulate the values of these subparts from within the language.

3.2 Language Concepts
The list of criteria formulated in the previous section is more or less a blueprint for an ideal programming language. In practice, however, some of the criteria are hard to combine. Examples of this are the criteria preservation of information and abstraction. The introduction of abstraction may cause some loss of information needed by the compiler to produce efficiently running codes.

To get more grip on the process of putting the criteria into practice, we will distinguish between a number of language concepts. From these concepts, we will take those that best adhere to the formulated criteria.
Language Concepts

Language concepts are those elements or properties of programming languages that distinguish them from others, such as the kind of data structuring mechanism, the programming paradigm (declarative or imperative), etc. Below, we list a number of (parallel) programming language concepts.

Data Type System

A data type system can be static as found in languages such as Fortran and C, but can also be dynamic, such as in Lisp or Prolog. A static system allows compile-time checking, whereas a dynamic type system potentially postpones checking to run-time, but allows more flexibility on the programming level.

Memory Model

For the memory model, we distinguish between two different properties:

- Single Assignment versus Memory Re-use
  In some languages a memory location may only be assigned a value once, whereas in others a location with a unique identifier is always reused when a value is re-assigned.
- Removal of dynamically allocated memory
  Dynamically allocated memory can be regained by performing implicit garbage collection during run-time or explicitly by the programmer.

Data Structures

For data structures one can incorporate arrays, lists, sets, queues, or provide an opportunity to the programmer to define customised data structures.

Access to Data

Data may be retrieved from memory directly (pointers) or indirectly by some memory model independent accessing mechanism (such as database-interfaces or arrays). In the first case, programmers have access to the underlying addressable memory, whereas in the latter case there is a mechanism that controls access.

Units of Computations

The units of computation [Brown86] may vary from expressions to statements, to functions (also procedures or subroutines), to tasks (modules). Encapsulated units of computation, such as functions and tasks may have several properties that are of importance. Functions may retain information between calls, functions may be defined recursively, they may have side-effects (global scoping), the parameters may have various modes of calling. In the lat-
ter case, actual parameters may adhere to the Church-Rosser property (that is multiple actual parameters of a function may be evaluated in parallel).

Definition of Repetition

There are two ways to define repetition: by some form of iteration (conditional jumps, and while- or for-loops) or by recursion, i.e., the repeated application of the same function.

Parallelism and Data Transport

Data transport can be defined explicitly with for example sends and receives, reads and writes, or implicitly, which is often closely linked to the definition of parallelism. For an explicit definition of parallelism, we can distinguish between data or task parallelism. Data parallelism induces multiple tasks by decomposing the data, whereas task parallelism defines multiple parallel (interacting) units of computation.

Declarative or Procedural Style

The most distinguishing factor between languages is the paradigm used for the description of a problem solution. In the procedural style one needs to specify exactly how to reach a solution, whereas in the declarative style one only specifies the what.

Mapping of Units of Computation

Units of computation can be mapped statically, dynamically, or adapted during run-time. Static mappings remain unchanged during computation. Dynamic mappings may change assignments of units of computations to processors in a predefined way. Adaptable mappings use parameters such as the load of the different processors to decide during computation where to assign units of computations to.

3.3 Influence of Language Concepts on the Design Criteria

Languages that adhere to certain language concepts allow for more successful translations than languages that do not. For a good language design focused on efficient translations, it is necessary to identify those constructs and combinations of constructs that translate successfully. We can identify grossly two reasons for bad translatability.

The first reason is the lack of necessary problem-information that disallows certain optimising transformations. This information is lacking because the programmer did not specify it or because the programming language has no method to describe it or by its nature forces the programmer to obscure it.
A second reason is the current state of compiler technology. If compilers would have sufficient knowledge about problem domains, and if pattern recognition would be native to these compilers, efficiency improving transformations could be feasible.

The first reason is the most troublesome, because no matter how compiler technology progresses more efficient transformations are not possible. We will now evaluate the language concepts listed in the previous section on their suitability for efficiency improving transformations.

Data Type System
A static data type system is preferable, because this allows for compile-time checking and saves run-time overhead. From the ease of programming point of view a dynamic data type system could be convenient, but scientific applications only use a limited number of data types.

Memory Model
Single assignment semantics may result in run-time copy problems of data structures, whereas memory re-use may result in loss of parallelism due to additionally induced data dependencies. Evidently, we want both possibilities and we should therefore leave the choice to the programmer.

Secondly, because implicit garbage collection induces — unpredictable — run-time overhead, explicit removal of dynamic memory is preferable.

Data Structures
Given the application domain the use of arrays is to be preferred, also because of the efficient implementations compared to for example lists.

Access to Data
The pointer access model is not selected because it causes problems in predicting the consequences of data access at compile-time. Shared-variables are not selected because they induce run-time system overhead. This leaves relative access of arrays via indices as the only option.

Units of Computations
Freedom in deciding which unit of computation to use is important from the scalability criterion point of view. If information is retained between calls of encapsulated units of computation, this will hamper analysis of parallelism in programs and consequently efficient translation. Side-effects (or global scoping) are not desired from the information hiding criterion point of view. Finally, the evaluation order of actual parameters of encapsulated units of computation must be free, to be able to extract parallelism from these calls.
Definition of Repetition

Repetition defined by recursion is difficult to analyse for potential parallelism. Only simple tail-recursions are resolvable at compile-time. Therefore, recursion as a way to define repetition is not desirable from an efficiency point of view. Repetition defined by conditional jumps is not considered good programming practice, whereas repetition through loops with variables may result in loss of parallelism and cause data dependence analysis problems. Consequently repetition is inherently difficult and whenever possible, we should strive to use loops with as little data dependencies as possible.

Parallelism and Data Transport

An explicit notion of data transport in a program may create a machine-dependence and an implicit notion of data transport may cause delays. The same holds for implicit or explicit parallelism. The first may not result in efficient running programs, whereas the latter may introduce machine dependencies. Hence, both parallelism and data transport should be implicit but a programmer should also be able to explicitly influence the translation process if desired.

Declarative or Procedural Style

A procedural style of programming is more favourable from the preservation of information point of view. The declarative style is advantageous from the ease of programming and parability point of view, because it leaves much of the execution order indeterminate allowing for more flexibility in the translation process.

Mapping of Units of Computation
We need static, dynamic, as well as adaptive mapping of units of computation.

3.4 Design Choices

The conclusion to be drawn from the interaction of language concepts with design criteria is that the language should have a set of sometimes contradictory concepts. We will therefore subdivide the group of selected concepts in two non-conflicting subgroups, one for a language describing the ideal algorithm, the other for a language influencing the mapping to and translation of that ideal algorithm.

The above choice relates to the algorithm development cycle depicted in Figure 3. First, we introduce the notion of a virtual machine. The virtual machine serves as an abstraction to map the ideal algorithm to. The ideal algorithm combined with the virtual machine after translation yields the virtual (parallel) program. In the next stage, we introduce the notion of the real
machine. The real machine is a model of an existing machine and to this machine one maps the virtual machine. A virtual machine typically has more resources (processors, memories) than a real machine and the resulting "two-stage" mapping between these machines is used for scalability reasons. One virtual machine can thus be mapped to several real machines that vary in the number of processors and/or memories.

Figure 4 Language concepts related to the algorithm development cycle

The subdivision of language concepts is as follows:

Language for Ideal Algorithm
The language concept choices for a programming language describing the ideal algorithm are:

- An array-type data structure as the basic data structure, where a strict separation between index- and data-domain of the array exists;
- A static data type system;
- A functional style of operations on the index-domain and a procedural style on the data-domain;
- Parallelism and data transport are implicitly defined in the language.
We will refer to this language for the *ideal algorithm* as the *Booster*\(^1\) language.

**Language for Machine Mapping**

The language concept choices for a programming language that describes the machine definitions and mappings are:

- Definition of a *virtual machine*;
- Import of models that describe the *real machine*;
- Explicit specification of *computation* - and *data-organisation*;
- Static, dynamic, as well as adaptive mappings between the *virtual* and *real* machine;
- All mappings are parameterized with respect to the problem size and machine-parameters.

We will refer to the latter language as the *annotation* language, because it augments information to the language with the purpose of facilitating the translation process.

**Translation of Booster**

The *Booster* program and its annotations are translated to a collection of programs for the processing elements and the appropriate communication and synchronisation primitives as defined by the programming model of the target machine. For this translation, we use the concept of Single Program Multiple Data (SPMD) code (see [Karp87]). In this model, each processor obtains the same code, but executes its own sequence of instructions depending on the processor-number. The advantage of this approach is the compactness of (parallel) code because every processor executes the same program.

When we relate the principle of parallel programming through annotations to explicit parallel programming with communication and synchronisation constructs, we obtain the form depicted in Figure 5. The programmer defines by annotations some mapping in the data-computation organisation "plane" (left figure). From this annotation the compiler generates a corresponding, semantically equivalent parallel program with communication and synchronisation (right figure). This concept illustrates the duality between annotations on the one hand and communication and synchronisation primitives on the other.

---

\(^1\) The name *Booster* has been chosen, because we hope that this language will boost both programming productivity and machine performance-utilisation.
Figure 5 a+b The principle of annotations and compilation

As depicted in the right figure, the compiler can derive a number of alternative SPMD-programs for a given Booster program and annotation. The programmer may limit the degrees of freedom of the compiler by supplying more detailed annotations. In this way the languages meet the requirement of Balance of Gain in Performance and Cost.

3.5 The Sequel

Chapter 4 introduces the Booster language, answering the question posed in Section 2.4: Can we define a language on basis of these concepts?”. Chapter 5 describes the annotation language and the principle of programming with annotations. The combination of Booster and its annotation language need to answer part of the second question of Section 2.4: What language concepts or combinations of language concepts are most favourable to the criteria parability and ease of programming?".

Chapter 6 discusses the semantic basis for Booster and the annotation language. Chapter 7 describes the translation of Booster programs to this semantic basis. Chapter 8 elaborates on the integration of annotations and Booster programs and the generation of resulting parallel programs. These three Chapters together provide an answer to the second part of the above question: "What language concepts or combinations of language concepts are most favourable to the criterion efficiency?"
Chapter 4
The Booster Language

In this chapter the Booster language is introduced in the form of a primer. The language is explained by describing its syntactic building blocks and illustrated by simple examples. The introduction of the language in this chapter is kept rather informal. For a formal syntax and lexical structure of the language the reader is referred to Appendix C. A formal treatment of the semantics is deferred to Chapter 6. Several algorithms from numerical analysis and linear algebra that constitute the heart of target applications are used to illustrate the expressiveness of the language.

4.1 Overview of the Language
The language concepts of Booster alone, as described in the previous chapter, do not constitute a language, but designing a programming language remains more or less an intuitive process. In the overview of the language we will relate the concepts to the elements of the language.

4.1.1 Domains, Shapes, and Views
On the language level Booster distinguishes between index and data domains. An index domain consists of integer-values, structured as finite, ordered index sets. An index set is a set of d-tuples with — non-negative — integer elements. In this context, d is the dimension of the index set.

A data domain consists of values of a certain data type. A data type in Booster is either a simple type, such as boolean, integer, or real, or is defined in nested form by an aggregated type record. The latter construct is not discussed in this thesis.

The distinction between index and data domain is reflected by the two fundamental concepts of the Booster language: shapes and views. The shape is the only data structure available comparable to data structures such as arrays or indexed files. Like an array, a shape is a finite set of elements with values of a specified data type, where each shape-element uniquely relates to an index from the shape-index set. So far, the shape does not deviate from the traditional array structure as found in other languages. The differences between an array and a shape are that a shape need not necessarily be rectangular and that the actual representation of a shape may be far more complex than the often linear representation of an array. We defer this issue of shape representation to
Chapter 5. In the sequel, we will refer to the set of values associated with the shape-elements as the content of the shape.

The second concept is that of the view. A view is a reference to selected shape elements. This selection defines a relation that takes the index set of a shape as input and returns a — combination of — selections, permutations, reshapings, or expansions of the input index set. A selection simply consists of filtering out a subset of the input index set. A permutation creates a different ordering on the indices of the input. A reshaping changes the dimension of the input index set, and an expansion creates multiple instances of the input index set.

The resulting selection of shape-elements can be 'viewed' again. The new view is defined on the output index set of the previously defined view, and so on. The functions that define the views can be composed to form more complex functions, provided that domains and co-domains of these functions match.

4.1.2 Programming Constructs

In imperative languages the manipulation of index values is not different from the manipulation of data values. Booster, however, strictly separates index manipulations from data manipulations. This is reflected in syntactically different constructs, namely the '<-' symbol for manipulations on index sets and the ':=' symbol for manipulations on data sets.

The view concept in Booster offers the programmer the possibility to abstract the access to indexed data structures. Instead of reasoning with loops and index variables, as found in imperative languages or function recursion in functional languages, the programmer defines references in Booster with views on shapes. Views (on shapes) are specified and modified in so-called view statements that assign a reference to a view identifier. For programming convenience, view statements can be encapsulated in view functions. A view function declaration contains a formal parameter list and a view function body, consisting of view statements. View functions can be used as functions in view statements, with (views on) shapes as actual parameters. View functions may be defined recursively.

To actually change the content of a shape, a content statement is available. A content statement replaces, much like the assignment statement found in imperative languages, the content of (a part of) a shape by an evaluated expression containing (views on) shapes and operators. Contrary to assignment statements in imperative languages where operations are usually scalar, operators in content statements may be multi-dimensional.

Booster supports the encapsulation of logically related content statements and view statements in functions. A function declaration contains a formal parameter list, local variable declarations, and a function body. The latter con-
sists of a sequence of view- and/or content statements. The difference with the
concept of a function found in PASCAL or C is that these statements may only
access the formal parameters or the locally defined variables of the function, in
other words: there are no side effects. Furthermore, values of the local variables
are never retained between two invocations: a function is "memoryless."

In Booster, two control structures for iteration and conditional execution are
available. For iteration the while-iter-statement is available. This statement
repeats its body until the specified condition fails or until it reaches the speci-
fied number of iterations. For conditional execution the if-then-else-statement
is available.

A Booster program is a collection of separately defined parts called modules.
Each module defines a number of operations and an interface to other modules
and its environment. The reader is referred to [Jong92] for Booster in the
ParTool environment.

4.2 Shapes and Views

4.2.1 Data Domains and Type Definitions

A data domain consists of a finite number of values of a unique data type.
Available data types are simple types, like boolean, integer, real, or a compound
type record.

Simple Data Types

The data domain of the type BOOL consists of the truth values {TRUE, FALSE}.
Both TRUE and FALSE are reserved words in Booster. Although all reserved
words are in capitals, the Booster language is not case-sensitive and the repre-
sentation {true, false} is also valid. The data domain of type INT consists of
integer-values, and the data domain of type REAL has real numbers of finite
precision. We will return to the issue of data types, their precision, and their
representation in the next chapter. To specify each of the simple data types the
reserved words BOOL, INT, and REAL are available.
We enclose all syntactic templates in boxes, following the convention that all
terminals (keywords) are in uppercase and all non-terminals in italic. In these
templates we use a set of symbols to describe optional or repeating syntactic
constructs. The following symbols and corresponding semantics are used:

{ x } Construct x appears zero or more times.
{ x }+ Construct x appears one or more times.
[ x ] Construct x is optional, and if included appears only once.
x | y Construct x or y is used.
4.2.2 Shape Declarations

The shape is the only data structure available in Booster. Shapes are — like arrays in other languages — declared as a finite set of elements with values of some specified data-type, where each shape-element relates to an index from the rectangular index set. The template of a declaration is as follows:

\[
\text{SHAPE \{ shape-id \ (, shape-id \}) (cardinality-specification) }
\]

\[
\text{OF type-specifier ; } )^+\]

The cardinality specification is a list of expressions containing integers, parameters, and integer-operators +, -, *, mod, and div. The cardinality symbol '\#' separates the expressions. The cardinality specification defines the index set of the shape, following the convention that the index sets of shapes always start from zero in every dimension. For example, a cardinality specification \((n \# m)\) yields an index set \(\{0,1,2,\ldots,n-1\} \times \{0,1,2,\ldots,m-1\}\), where \(\times\) denotes to Cartesian product.

Consider the following shape declarations:

\[
\text{SHAPE } V, W \ (n) \text{ OF REAL; }
\]

\[
A \ (m \# 2*n) \text{ OF INT;}
\]

Both \(V\) and \(W\) are declared as a vector of length \(n\) with data type \texttt{REAL}. \(A\) is declared as a \(m \times 2*n\) matrix of the data type \texttt{integer}. The parameters \(n\) and \(m\) used in this example are part of the context of the shape declaration. An example of such a context is a (view)function or module.

4.2.3 Index Domains and Index Set Definitions

The index domain consists of index sets that are finite sets of \(d\)-tuples with non-negative natural numbers as elements. Booster has built-in functions for manipulating index sets and functions that return some property about index sets. First we consider functions that construct and manipulate index sets.

Index Set Operators

Table I lists the available index set manipulation operators with their priority. The first operator is the range operator represented by a colon '\:' . The range operator takes two non-negative integers as arguments and returns a set of elements \((a, b \in \{0,1,2,\ldots\}, a \leq b)\):

\[
\text{a:b = \{ a, a+1, a+2, \ldots, b \}}
\]

When \(b\) is smaller than \(a\), the result of the range operator is empty, which is denoted as \(\varnothing\). The union operator, represented by a bar '\l\', takes two index sets as arguments and returns the union. Examples of this operator are:
4.2 SHAPES AND VIEWS

\[ \begin{align*}
1:5 & \mid 3:8 \equiv 1:8 \\
2:5 & \mid 1:5 \equiv 1:5
\end{align*} \]

The \textit{intersect} operator, denoted by an ampersand \('\&'\), takes two index sets as arguments and returns their intersection. Examples are:

\[ \begin{align*}
3:4 \ &\& \ 4:7 & \equiv 4 \\
6:8 \ &\& \ 4:7 & \equiv 6:7 \\
1:5 \ &\& \ 8:12 & \equiv \emptyset
\end{align*} \]

The last operator is the \textit{difference}-operator, represented by the backslash '\(\backslash\)', as shown below:

\[ \begin{align*}
1:5 \ \backslash \ 3:8 & \equiv 1:2 \\
2:5 \ \backslash \ 1:7 & \equiv \emptyset
\end{align*} \]

Another appearance of the difference operator will be discussed in Section 4.2.4.

\textbf{Table I} Operators for constructing and manipulating index sets

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>:</td>
<td>5</td>
</tr>
<tr>
<td>union</td>
<td>|</td>
<td>2</td>
</tr>
<tr>
<td>intersection</td>
<td>&amp;</td>
<td>3</td>
</tr>
<tr>
<td>difference</td>
<td>\ \backslash</td>
<td>4</td>
</tr>
</tbody>
</table>

Each operator has an associated \textit{priority} to resolve syntactic ambiguities for expressions with more than one operator. For example, consider the following expression:

\[ 2:3 \ \backslash \ 3:4 \ &\& \ 4 = \begin{cases} 
(2:3 \ \backslash \ 3:4) \ &\& \ 4 & = 2 \ &\& \ 4 = \emptyset \\
2:3 \ \backslash \ (3:4 \ &\& \ 4) & = 2:3 \ \backslash \ 4 = 2:3 
\end{cases} \]

Two alternative evaluation orders exist as is demonstrated above. By using the priorities of operators \(\backslash\) and \&, we can resolve the order: operator \(\backslash\) has a higher priority (4) than \& (3) and therefore the first alternative is the correct interpretation. Priorities of operators range from 1 to 10, where only monadic operators have the highest priority.

\textbf{The size function}

The \textit{size} function is available to retrieve the size of index sets of shapes. This function takes the index set of a shape as argument and returns the its size, i.e. the number of elements in the index set. Examples of this function are given in Section 4.3.2.
4.2.4 Simple Views

Simple View Functions

The most simple form of a Booster view is \([\nu]\). \(\nu\) can be an expression, index set, or view function application. A simple view takes the index set of its argument as input and returns a selection, permutation, or reshaping of that index set. Consider the following application of the view \([\nu]\) to shape \(A\), with \(\nu\) equal to the index set \(1:5\).

\[
\text{SHAPE A (10) OF INT;}
\]
\[
\ldots A[1:5] \ldots
\]

The above view, when applied to \(A\), selects the shape-elements with indices 1 to 5. For example, we denote the selection of the second and fifth element as\(^1\):

\[
\ldots A[1:4] \ldots
\]

The following simple view function selects all but the fifth index or sixth element of \(A\).

\[
\ldots A[\backslash5] \ldots
\]

Note that the difference operator \(\backslash\) is the only operator used in a monadic context in simple view functions. This operator implicitly takes the index set of the structure on its left side as left argument.

It is also possible to use expressions as arguments for simple view functions. These expressions contain constant integer values, parameters, and integer operators like \(+\), \(-\), \(*\), \(\mod\), and \(\div\). Table II lists these operators and their priority.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>(___)</td>
<td>6</td>
</tr>
<tr>
<td>subtraction</td>
<td>(-)</td>
<td>6</td>
</tr>
<tr>
<td>multiplication</td>
<td>(*)</td>
<td>7</td>
</tr>
<tr>
<td>integer divide</td>
<td>(\div)</td>
<td>7</td>
</tr>
<tr>
<td>modulo</td>
<td>(\mod)</td>
<td>7</td>
</tr>
</tbody>
</table>

Examples of expressions in simple views are:

\[
A[n \div 4]
\]
\[
A[1:n+3]
\]
\[
A[n-6:n*3 \ p:(p+1)]
\]

(interpreted as \(A[1:(n+3)]\))

(interpreted as \(A[((n-6):(n*3)) \ (p:(p+1))])\)

\(^1\) Note that a shape of cardinality 10 has indices numbered from 0 to 9. The second and fifth element therefore have index 1 and 4, respectively.
4.2 SHAPES AND VIEWS

Intrinsic Parameters

*Booster* has a number of intrinsic parameters for use in simple view function applications. Table III lists these parameters, the *lowerbound*, *upperbound*, and *dimension selection*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowerbound</td>
<td>lwb</td>
</tr>
<tr>
<td>Upperbound</td>
<td>upb</td>
</tr>
<tr>
<td>Dimension Selection</td>
<td>_</td>
</tr>
</tbody>
</table>

These intrinsic parameters serve to define simple view functions that depend on their argument. For example, the selection of the last two elements of a shape becomes:

```
SHAPE V (50) OF INT;
... V[upb-1:upb] ...
```

This view, when applied to \(V\), is equivalent to \(48:49\). If, on the other hand, it is applied to the following shape \(w\),

```
SHAPE W (100) OF INT;
... W[upb-1:upb] ...
```

the view \([upb-1:upb]\) is equivalent to \(98:99\). The dimension selection is a placeholder for the range \(lwb:upb\), e.g.:

```
SHAPE A (n ≠ m) OF INT;
... A[_,4:5] ...
```

This view selects the fifth and sixth column of the matrix \(A\).

Simple View Compositions

We can compose simple views to form more complex views. This composition results in an intersection of all simple views used. Consider,

```
SHAPE U (12) OF INT;
... U[2:6][3:5][upb-1:upb] ...
```

The leftmost simple view results in the selection of the indices \(2,3,4,5,6\). The application of the next simple view \([3:5]\) results in the set \(2,3,4,5,6 \cap 3,4,5\) = \(3,4,5\). The upperbound of this last set is 5 and hence the whole composition is equivalent to \(U[4:5]\).

The intersection of views evaluates from left to right and therefore the domain of the view on the right uses the resulting index set of the expression on the left as input:
\[ U[\emptyset] = \text{error} \quad \{ \text{empty set} \} \]
\[ U[1:10][2:3][1:5] = \text{error} \quad \{ \text{index out of range} \} \]
\[ U[1:10][1:5][2:3] = U[2:3] \]

The intended resulting view can also be obtained using intersection of the index sets:
\[ U[1:10 \& 1:5 \& 2:3] = U[2:3] \]

A more intricate composition of simple views in the multi-dimensional case is the so-called real selection. The dimension of \( A[1,\_] \) equals 1, e.g.

\[
\text{SHAPE } A \ (10 \ # \ 10) \ \text{OF REAL;} \\
\ldots \ A[1,\_][2:8] \ldots
\]

The first simple view selects the second row of the matrix \( A \) and implements a real selection because it yields a row (vector) instead of a matrix. This row is a one-dimensional structure that permits the application of a view \( [2:8] \). As a result this composition selects the third to ninth element of the second row of \( A \), i.e. \( \{ A[1,2], A[1,3], \ldots, A[1,8] \} \).

### 4.3 View Operations

The separation between operations on the index and data domain, characteristic for Booster, manifests itself in separate language constructs for operations on index sets. One such language construct is the view statement, which allows for the definition and manipulation of so-called view identifiers. The view function serves as a construct to encapsulate these view statements.

#### 4.3.1 View Statements

A view statement defines views and identifies them by assigning the result to a view identifier. These view identifiers are again used in other view statements or in content operations (the latter are discussed in Section 4.4). Through a view identifier, like a view, it is possible to access the elements of the shape it refers to. It is important to observe that view identifiers, unlike shapes, are not persistent. They may be subject to optimisation and normally the translation process eliminates them.

View statements incorporate the ‘<-’ or draw from symbol to denote the binding of view identifiers to shapes. In view statements, a view identifier is assigned the view expression on the right side of the ‘<-’ symbol. For example:

\[
\text{SHAPE } A \ (3) \ \text{OF INT;} \\
V <- A;
\]

the view identifier unambiguously without requiring any additional declaration. This definition remains valid until the program ends or until a redefinition of the view identifier in a succeeding view statement occurs.

A last important observation is that view identifiers are treated as shapes when used in any other context than the left side of a view statement. Therefore, the same restrictions on index sets for shapes are valid for index sets for view identifiers: they must be consecutive, rectangular, and the index must start from zero in every dimension.

Before introducing the kind of relations defined by view statements, we elaborate on two preliminary concepts. First, we introduce the concept of renumbering induced by view statements. Second, we elaborate on syntactic constructs used for defining relations between view identifiers and shapes.

**Renumbering**

View statements have one important effect when assigning views to view identifiers: the index set of the view identifier is renumbered from 0 onwards in every dimension. For example,

```plaintext
SHAPE A (3 # 4) OF INT;
V <- A[1:2,2:3];
```

This view statement defines a view identifier \( V \) with cardinality \( 2 \times 2 \) (derived from the cardinality of the right argument \( A[1:2,2:3] \)). The view identifier’s index set equals \( (0,1) \times (0,1) \) (a renumbering of the index set of \( A \)). Consequently, the view statement defines a relation between \( V \) and \( A \), where \( V[0,0] \) refers to \( A[1,2] \), \( V[0,1] \) refers to \( A[1,3] \), \( V[1,0] \) refers to \( A[2,2] \), and \( V[1,1] \) refers to \( A[2,3] \). Any initialised view identifier \( V \) can in turn be used in other view statements, e.g.

```plaintext
V <- V[0,0:1];
```

The last statement redefines the view identifier \( V \) to refer to the first row of the previous \( V \). In terms of \( A \) it refers to \( A[1,2:3] \).

**Free Variables**

For more elaborate (view) relations Booster uses the free variable construct. A free variable can occur in view- and content statements and is bound to the index set of the argument it is applied to. The scope of the free variable is always limited to a view or content statement. For example:

```plaintext
SHAPE P (11) OF INT;
V <- P[lwb+1:upb-1];
L[i] <- V[3*i];
```
The first view statement defines view identifier \( V \), which selects all but the first and last element of \( P \). After renumbering, \( V[0] \) refers to \( P[1] \), \( V[1] \) to \( P[2] \), etc. The second statement defines a relation between view identifier \( L \) and \( V \) using the free variable \( i \). The relation between \( L \), \( V \), and \( P \) defined by both view statements is depicted in Figure 6.

![Diagram showing relation between view identifiers L, V, and shape P.](image)

**Figure 6. Relation between the view identifiers \( L \) and \( V \) and shape \( P \).**

The second view statement defines a relation between \( L \) and \( V \) such that \( L[0] \) refers to \( V[0] \), and \( L[1] \) refers to \( V[3] \), etc., selecting one in every three elements of \( V \). Here, the function \( 3*i \) and the index set of \( V \), \( \{0,1,...,8\} \), determine the range of the free variable \( i \). This results in the condition \( 3*i \in \{0,1,...,8\} \) from which we can derive the — consecutive — range \( \{0,1,2\} \) for \( i \).

**Rules**

A restriction on the use of free variables in view statements is that all free variables must occur free on the left side of the '<=' symbol. The rationale is that if free variables do not occur free on the left side of the '<=' symbol, non-consecutive index sets for free-variables could result. Consider for example an — incorrect — view statement \( V[2*i] \leftarrow P[i] \), which would result in an index set \( \{0,2,4,6,8\} \) for \( V \). This index set is not consecutive, and hence not permitted in Booster.

The use of several distinct free variables in one view statement is also possible. Consider:

```plaintext
SHAPE A (10 # 20) OF INT;
V[i,j] <- A[2*i,2*j];
```

where \( i \) and \( j \) are bound to \( 0:4 \) and \( 0:9 \), respectively. Free variables may occur in expressions in combination with integer expressions, operators +, -, *, div, and mod, and index expressions.
4.3 VIEW OPERATIONS

For view statements, we distinguish between different types of relations. These different types, selection-, permutation-, expanding-, and reshaping relations illustrate the syntax and semantics of the view statement.

Selection Views

Many algorithms define computations on subsections of shapes. To define these subsections Booster uses selection views. For example:

```
SHAPE A (n # m) OF INT;
V <- A[0:1, _];
```

The view statement redefines the view identifier \( V \) to refer to a selection of \( A \), namely the first two rows of \( A \). \( V \) now has cardinality \( 2 \times m \) and the derived index set is: \( (0,1) \times (0,1,2,...,m-1) \).

Permutation Views

Other algorithms need views that rearrange or permute the index set of a shape. In this case we typically use the free variable, as is illustrated below:

```
SHAPE W (20) OF REAL;
V[i] <- W[19-i];
```

Here, \( V[0] \) refers to \( W[19] \), \( V[1] \) to \( W[18] \), etc. An example of a rotation is:

```
V[i] <- W[(i+1) mod 20];
```

This view statement intends to implement a left rotate of one element. There is, however, an ambiguity, because the exact range of \( i \) cannot be deduced from this statement. It requires solving \( i \) out of the relation \((i+1) mod 20 \in 0:19\). For example, ranges like \( 0:19 \), \( 0:39 \), ... are solutions to this inequality. The resulting inequalities \( 0 \leq (i+1) mod 20 \leq 19 \) have an infinite number of integer solutions and it is not clear which one to select.

Index domain specifications

To overcome such a problem, Booster has a so-called index domain specification which specifies a range for the free variable(s), for example:

```
V (20) [i] <- W[(i+1) mod 20];
```

The expression between curly bracket specifies the intended index domain. Multi-dimensional index domains separate each dimension by the dimension separator ‘#’. For example:

```
SHAPE A (n # n) OF REAL;
M { n div 2 # n } [i, j] <- A[2*i, (i-1) mod n];
```
Here, a selection is defined on the first dimension and a permutation on the second. As demonstrated, the index domain specification may contain expressions involving parameters, integers, and integer operators.

Reshaping Views
The third type of relation is reshaping. Reshaping is used to view a shape in a different dimension. For example, a relation between a vector \( v \) and a matrix \( M \):

\[
\text{SHAPE} \ v \ (n) \ \text{OF} \ \text{INT};
M \ (n \ \text{div} \ 10 \ # \ 10) \ [i,j] \ <- \ v[i*10 + j];
\]

The index domains for \( i \) and \( j \) are \( 0:(n \ \text{div} \ 10)-1 \) and \( 0:9 \), respectively. This so-called \textit{dimension-increasing} view establishes the following relation:

\[
M[0,0] = v[0] \quad M[1,0] = v[10] \quad M[2,1] = v[21], \text{ etc.}
\]

This type of view plays an important role in the introduction of parallelism. We will elaborate on this in the next chapter. Apart from dimension increase, reshaping can also define dimension decrease. View a matrix \( B \) as a vector \( v \) for example:

\[
\text{SHAPE} \ b \ (n \ # \ m) \ \text{OF} \ \text{INT};
v \ (n*m) \ [i] \ <- \ b[i \ \text{div} \ n, \ i \ \text{mod} \ n];
\]

This \textit{dimension decreasing} view relates \( v \) and \( b \), such that \( v \) is a row-wise linearisation of \( b \). Figure 7 depicts the relation between \( v \) and \( b \).

\[
\begin{array}{c}
\text{Figure 7 Dimension decreasing view.}
\end{array}
\]

Expanding Views
The three previous relations define one-to-one relations between index sets of view identifiers and shapes. The last type — the \textit{expanding} view — defines a one-to-many relation. Multiple elements of the view identifier may relate to the same element of the shape. An example:

\[
\text{SHAPE} \ a \ (6) \ \text{OF} \ \text{INT};
M \ (4 \ # \ 3) \ [k,i] \ <- \ a[k + i];
\]
Figure 8 depicts this view, where all elements of \( M \) on the same — dashed — diagonal line refer to the corresponding element of \( A \) on this diagonal. Consequently, \( M[2,0] \), as well as \( M[1,1] \), and \( M[0,2] \) all refer to \( A[2] \).

![Figure 8 An expanding view on shape A resulting in view identifier M.](image)

The usefulness of these views will be demonstrated in a number of examples in Section 4.6.

**Summary**

Summarising, view statements define various types of relations between view identifiers and — eventually — shapes. View identifiers, however, cannot identify arbitrary subsets of shapes. Selected index sets must adhere to the following rules:

- View identifiers can only represent finite selections;
- The subsets of shapes referred to by view identifiers must be collapsible to rectangular index sets;
- Index domains must be specified in view statements when a non-linear function or a dimension increasing view is used.

The last point needs some additional explanation. Because in some cases ambiguities arise, the language requires the specification of index domains for view statements. In other cases, however, the correct index domain can be derived from the involved view functions and an additional specification of the index domain by the programmer is superfluous. Therefore, the language offers the option to omit these non-ambiguous index domain specifications.

**4.3.2 Content Selection Views**

The above types of views use selections that are independent of the shape's content. In this section, we address the so-called content selection views. This view is a simple view function where the argument is either a Boolean expres-
sion, a selection function, or an index expression containing index sets and Boolean expressions.

The discussion of content selection views is organised as follows. We start with an interlude which introduces — multi-dimensional — Boolean expressions. Next, we discuss content selection views, but separate one- and multi-dimensional cases, because these are fairly different from a language point of view.

**Boolean Expressions**

Boolean expressions in Booster consist of expressions involving (views on) shapes and comparison operators. The comparison operators are listed in Table IV, together with their priority and representations. The interpretation of each operator is self-explanatory.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>5</td>
</tr>
<tr>
<td>less than or equal to</td>
<td>&lt;=</td>
<td>5</td>
</tr>
<tr>
<td>equal to</td>
<td>=</td>
<td>5</td>
</tr>
<tr>
<td>not equal to</td>
<td>&lt;&gt;</td>
<td>5</td>
</tr>
<tr>
<td>greater than or equal to</td>
<td>&gt;=</td>
<td>5</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>5</td>
</tr>
</tbody>
</table>

These operators also take multi-dimensional data structures as arguments, which results in an element-wise application of the operator. A more formal description is given in Chapter 7.

**One-dimensional Content Selection Views**

Multi-dimensional content selection views introduce some additional problems and we will therefore consider the more simple one-dimensional case first to illustrate the principle. Consider the following content selection view:

```plaintext
SHAPE A (5) OF INT;
B (4) OF REAL;
... A[B > 1] ...
```

The semantics of this content selection view \( [B > 1] \) is as follows: The Boolean expression \( B > 1 \) yields an index set \( I \) with elements for which the corresponding values are larger than one. The simple view function \( [B > 1] \) selects those indices of \( A \) for which the condition is valid. For example, consider the following content of \( B \):

\[
B = \{0.0, 0.5, 1.1, 2.0\}
\]
Given the condition \( B > 1 \) the view selects the indices \([3, 4]\). consequently the view applied to \( A \) selects \( A[3] \) and \( A[4] \). Note that content selection is valid despite the difference in size between \( A \) and \( B \).

The language allows for more complex Boolean expressions, such as:

\[
\text{SHAPE C, D (2) OF REAL;} \\
... A[C > B[3:4]] ...
\]

It is important to note that the selected index set is derived from the left argument of the Boolean expression. For example, the first view selects those elements of \( B[3:4] \) which are larger than the corresponding element of \( C \). Consequently, the selected index set is a subset of \([3,4]\) and not \([0,1]\). The second view, however, selects a subset of \([0,1]\) and not \([3,4]\), because \( C \) is the left argument now.

**Index Operators**

We can also use multiple Boolean expressions in combination with index set operators (see Table I, Section 4.2.3). Examples are given below:

\[
... A[ A > 0 & B < 2] ... \\
... A[ 1:4 | C < D] ...
\]

The first view selects the index set that satisfies \( A > 0 \) and the index set that satisfies \( B < 2 \) and intersects both sets. The second view in the example selects the union of the index set \([1:4]\) and the index set selected by \( C < D \).

The size function, which was already mentioned in Section 4.2.3, can, apart from shapes, also be applied to viewed shapes. For example, 

\[
\text{size}(A[ A > 0])
\]

yields the number of elements of \( A \) that are larger than zero.

**Abbreviations**

For convenience, the programmer can make use of a syntactic shortcut to specify certain content selection views. For example the following views are equivalent:

\[
A[A > 0] \equiv A[ > 0] \\
\]

In general, content selection views yield a — possibly non-consecutive — structure of which the number of elements is only known at run-time. It is therefore unfeasible to check the correctness of certain content statements at compile-time. This is illustrated by the following example:
SHAPE A, B (n # m) OF REAL;

The content statement is only correct when A[>3], A[>0], and B[>2] select corresponding elements: in all other cases a run-time error will be generated.

Multi-dimensional Content Selection Views

The use of content selection views in view statements involving multi-dimensional structures is more complicated. Recall that in view statements the selected indices are renumbered when assigned to view identifiers. This means that when assigning multi-dimensional content selections to view identifiers a problem arises. Consider the following program fragment:

\[ V \leftarrow A[>0]; \]

where A is a previously declared \( n \times m \) matrix. The required renumbering for view statements resulting in rectangular index sets is not always possible in this case. To illustrate this point, assume for example that A[>0] selects the index set \( \{ (1,1), (1,2), (4,1) \} \). This set cannot be renumbered to form a rectangular, consecutive index set.

Rules

Because arbitrary selections of multi-dimensional structures cannot be renumbered to form a rectangular, consecutive index set, the use of multi-dimensional content selection views in view statements is not allowed.

A second restriction on the use of content selection views is the combination with other selections within one simple view function, because this may result in ambiguous or incomprehensible constructs. An example is the following:

\[ A[0:2,>0] := \ldots; \]

The view selects the first 3 rows of the first dimension and all elements larger than zero for second dimension. The two following — distinct — meanings can be associated with the above construct:

\[ A[0:2,\_] [>0] := \ldots; \]
\[ A[>0][0:2,\_] := \ldots; \]

Because of this ambiguity, the language does not permit the use of content selection views in conjunction with other views in one simple view function.

4.3.3 View Statements and Index Domains

We now come to the point where we can present the general template of a view statement. Up to now the view statements contained at most one index
domain specification on the left. The general construct however, has two domain specifications:

\[
\text{view-identifier} \ ['\text{cardinality-spec}'] ['\text{free-variable-list}'] \leftarrow \\
\text{content-expression} \ ['\text{cardinality-spec}'] ['\text{free-variable-expr}']
\]

A restriction on this syntax template is that the last cardinality specification may only contain variables that occur free. An example of this general statement is:

\[
\text{SHAPE A (n) OF REAL;}
\text{V \{ n \ div \ 2 \} [i] \leftarrow A[i \ mod \ 1000];}
\text{V \{ m \ div \ 2 \} [i] \leftarrow A \{ m \} [i \ mod \ 1000];}
\]

The two view statements have the same effect, but differ with respect to the domain specifications. The first uses the parameter \( n \) from the declaration of shape A to define the domain. The second view statement uses an extra domain specifier on the right to determine the size of the index domain of A. The index domain on the left: \{ m \ div \ 2 \} reuses the (new) domain variable \( m \).

We can best explain the rationale for incorporating the inclusion of a second index domain specification with an example:

\[
\text{V \{ ... ?? ... \}[i] \leftarrow A[\text{some_function}][2*i];}
\]

A function \text{some_function} views a structure A and after selection of all even elements assigns the result to a view identifier V. Here, we do not know the index domain returned by the view function \text{some_function}, hence we cannot determine the index domain on the left.\(^2\) To overcome this, we need the aforementioned specification of the index domain on the right side of the \('\leftarrow'\) symbol. In this case, it would yield:

\[
\text{V \{ n \ div \ 2 \}[i] \leftarrow A[\text{some_function}] \{ n \} [2*i];}
\]

The size of the index domain returned by A[\text{some_function}] is bound to the domain specification \{ n \} and used on the left of the view statement in the domain specification \{ n \ div \ 2 \}. The multi-dimensional case for these domain specifications is straightforward:

\[
\text{SHAPE B (a \# b) OF INT;}
\text{V \{ n \ div \ 2 \# 2*m \} [i,j] \leftarrow A[\text{some_function}] \{ n \# m \} [2*i,j \ mod \ n];}
\]

### 4.3.4 View Functions

View statements in Booster can be encapsulated in view functions. A view function defines an interface through a formal argument list with a clear dis-

\(^2\) The view function \text{some_function} may be data dependent and therefore no domain can be specified in general.
tinction between input- and output-arguments. The input arguments — enclosed by parenthesis — are on the left of the '->' sign and the output arguments on the right. The arguments come in two flavours, (viewed) shape or index arguments. The functions opens with the declaration of the cardinality of both the input and output shape arguments. The cardinality of the output arguments uses the cardinality of the input shape arguments or a constant expression. The body of the view function consists of one or more view statements, optionally using previously defined view functions.

The template of a view function declaration is:

\[
\text{VIEW FUNCTION view-function-id ( arg \{, arg \})-\rightarrow ( arg \{, arg \});} \\
\text{\{arg-def;\} }^+ \text{BEGIN} \\
\text{\{view-statement;\} }^+ \text{END;}
\]

We illustrate the definition of the cardinality of the formal arguments with an example declaration of a view function:

\[
\text{VIEW FUNCTION Filter (A, k) -\rightarrow (B);} \\
\text{A (n); B (n div k);} \\
\text{BEGIN} \\
\text{B[i] \leftarrow A[k*i];} \\
\text{END;}
\]

The view function Filter has an input shape argument \( A \) of cardinality \( n \) and an index argument \( k \). Furthermore, the shape output argument \( B \) has a cardinality \( n \text{ div } k \) defined with the index argument \( k \) and parameter \( n \). In the body of the view function, a view statement defines the relation \( B_i = A_{k*i} \) between \( A \) and \( B \), e.g.:

\[
\text{SHAPE C (10) OF INT;} \\
\text{W \leftarrow filter(C,2);} \\
\]

In the view statement the application of view function Filter to shape C with index argument 2, results in the definition of view identifier \( W \) for which \( W[i] \) refers to \( C[2*i] \).

Rules

Before we present more examples of view functions, we must define more precisely the syntactic and semantic conventions. The first convention is concerned with the use of shape input and output arguments in the view function body. The view function is to be seen as a more complex simple view that defines some relation from shape input to shape output arguments. Therefore,
shape input arguments may only be used on the right side of any view statement in the view function body, whereas output arguments may only be used on the left side of any view statement. A sound and comprehensible relation between input and output can then be established.

Index arguments, however, intend to parameterize the relation defined between shape arguments and may only occur in the input argument list and not in the output argument list. Furthermore, index arguments always represent some index value and there is no need to declare them. Moreover, index arguments may occur in the scope of the function body and in the cardinality definition of the shape output arguments. In an actual view function call in the program, the arguments are restricted to integer expressions. These integer expressions may contain free variables and parameters.

More Advanced View Functions

A typical application of a view function is the definition of several interrelated simple view functions that yield multiple output arguments, e.g.

```plaintext
VIEW FUNCTION Unzip (V) -> (Even, Odd);
V (n); Even (n div 2); Odd (n div 2 + n mod 2);
BEGIN
   Even <- Filter(V,2);
   Odd <- Filter(V[1:upb],2);
END;
SHAPE A (11) OF REAL;
(Even_elements, Odd_elements) <- Unzip(A);
```

The view function Unzip uses the previously defined function filter to select all even and odd elements of a certain vector v. The application of the function Unzip to a shape A returns even and odd elements of A and assigns them to the actual arguments, Even_elements and Odd_elements, respectively.

The view functions considered so far take some set of shapes as input and return a view on these shapes, regardless of their contents. These kind of views are the so-called structural view functions. Below we illustrate the use of content-dependent view functions, for example:

---

3 The cardinality definition of the formal data input arguments effectively defines an interface to the cardinality of the actual parameter to which the function is applied. Here, the use of an index argument is inept.
VIEW FUNCTION Split (A, Element) -> (Small, Equal, Large);
A (n); Element (1); Small (s); Equal (e); Large (l);
BEGIN
  Small <- A[< Element];
  Equal <- A[= Element];
  Large <- A[> Element];
END;

This view function Split takes some vector A and a scalar Element as input
and returns the elements that are either smaller, equal, or larger than the —
content of the — scalar. Depending on the content of A and Element, the three
output arguments may have any cardinality.

Empty views
Views that define relations using content selection views can yield so-called
empty views. For example, consider the view function Split which returns an
empty view Large if no value of A is larger than Element. Any operation on an
empty view — except for the size function — will always result in an error con-
dition. The size function will return zero if a view is empty.

View functions in simple views
A notation within the framework of simple view functions facilitates the use of
view functions. Consider the following two equivalent view statements:

SHAPE A (n) OF REAL;
V <- Filter(A,2);
V <- A[Filter(2)];

The second view statement differs from the first statement in that the first
input shape argument of filter is not specified and is replaced by the —
implicit — argument of the simple view function. This notation, however, is
only allowed for view functions that have one output argument. The expres-
siveness of this syntactic construct is illustrated below by another example:

VIEW FUNCTION Diagonal (X) -> (Y);
X (n # n); Y (n);
BEGIN
  Y[i] <- X[i,i];
END;

SHAPE A (n # n # n # n) OF INT;
W <- A[_,Diagonal,Filter(2)];

The view function diagonal takes a two-dimensional argument as input and
returns a one-dimensional argument referring to the main diagonal of the
input. Both view functions are applied on different dimensions of the shape \( A \), resulting in a dimension-wise selection of a three-dimensional view identifier.

4.4 Content Operations

4.4.1 Content Statements

A content statement, much like the assignment statement found in most imperative languages, replaces the contents of a number of (viewed) shapes by an evaluated expression. This expression can contain shapes, view identifiers, view functions, functions, or operators. The `:=` or becomes symbol is used to denote the content assignment, where the expression on the right part of the becomes symbol is assigned to the left part, e.g.

\[
\text{SHAPE A, B (n) OF REAL;}
A := B;  
\]

The first content statement defines an element-wise assignment of the shape \( B \) to shape \( A \). Content statements in Booster can define assignments on any multidimensional structure that eventually refers to a shape. For example consider,

\[
\text{SHAPE A, B (n) OF REAL;}
C (n \neq m) OF REAL;  
C[_,1] := A;  
A[\text{\text{filter}(2)}] := B[\text{\text{filter}(2)}];
\]

The effect of the first content statement is the element-wise assignment of a selection of \( B \), indexed by \( \{2,3,4,5\} \), to the selection of \( A \), indexed by \( \{1,2,3,4\} \). This first statement illustrates the assignment of two vectors. In the second statement the first column of \( C \) is replaced by the vector \( A \). The last statement uses the view function \( \text{\text{filter}} \) — declared in Section 4.3.4 — to define the assignment of all even elements \( B \) to all even elements of \( A \).

Collateral Assignments

View functions can have more than one output argument, as was demonstrated by the view function \( \text{\text{Unzip}} \) from Section 4.3.4. These view functions can also be used in content statements resulting in the collateral assignment to shapes. Consider,

\[
\text{SHAPE Even, Odd (n) OF REAL;}
V (2*n) OF REAL;  
(Even, Odd) := \text{\text{Unzip}}(V);
\]
The view function \texttt{Unzip} is applied to \texttt{V}, resulting in an unraveling of the input and a collateral assignment to the shapes \texttt{Even} and \texttt{Odd}. The assignment may also be defined the other way around, which results in the collateral initialization of the shape \texttt{V}.

\begin{verbatim}
Unzip(V) := (Even, Odd);
\end{verbatim}

The general template is:

\begin{verbatim}
content-stat := view := content-expr | (view {, view}) := (content-expr {, content-expr})
\end{verbatim}

**Non-determinism**

The collateral statement can lead to non-determinism if the regions that are referred to by the different (viewed) shapes on the left side of the assignment, overlap. The use of these overlapping regions in collateral statements is therefore not permitted. Violation of this restriction is checked at run-time.

### 4.4.2 Standard Operators and Functions

In the preceding sections, different language constructs have been illustrated with content statements that modify (the content of) shapes. In this section, we enrich the vista of (content) expressions by introducing the set of standard — multi-dimensional — operators and functions. Prior to their introduction, we need to define an \textit{ordinal} function that will be used in the various definitions. The ordinal function or \texttt{ord} takes an \textit{index set} and a \textit{position} — an integer ranging from one to the size of the index set — as an argument and returns the index that corresponds with this position. The ordinal is obtained by assigning a relative number to an enumeration of the index set that starts from the first dimension onwards. The \texttt{ord} function is denoted as \texttt{ord(I,p)}, where \texttt{I} is the index set and \texttt{p} the relative number. This function is formally defined in Chapter 6 and is used here for illustration purposes.

**Operators**

We distinguish four types of operators: \textit{arithmetic}, \textit{comparison}, \textit{relational}, and \textit{user-defined} operators. \textit{Arithmetic} operators, such as addition and multiplication, are used in content expressions. The \textit{comparison} operators, like "greater than" and "equal to", are used in Boolean and selection expressions. The \textit{relational} operators are used to combine several Boolean or selection expressions. The introduction of the \textit{user-defined} operators is deferred to Section 4.4.4.

**Arithmetic Operators**

Table V gives the representation and priorities of the basic arithmetic operators.
4.4 CONTENT OPERATIONS

Table V Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>monadic plus</td>
<td>+</td>
<td>10</td>
</tr>
<tr>
<td>monadic minus</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>exponentiation</td>
<td>**</td>
<td>8</td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>7</td>
</tr>
<tr>
<td>real division</td>
<td>/</td>
<td>7</td>
</tr>
<tr>
<td>integer division</td>
<td>div, /</td>
<td>7</td>
</tr>
<tr>
<td>modulo</td>
<td>mod, %</td>
<td>7</td>
</tr>
<tr>
<td>dyadic plus</td>
<td>+</td>
<td>6</td>
</tr>
<tr>
<td>dyadic minus</td>
<td>-</td>
<td>6</td>
</tr>
</tbody>
</table>

Examples of content statements containing these operators are:

SHAPE A, B, C (n) OF REAL;
D (2*n) OF INT;
A := B*C / D[filter(2)];
(A, B) := Unzip(D*2);

The priorities of operators are again used to derive a correct evaluation order.

Comparison and Relational Operators

The comparison operators have already been introduced in Section 4.3.2 in Table II. The relational operators, for which the operands must be of the data type BOOL, are listed in Table VI. In this table, the not operator is monadic, the others are dyadic.

Table VI Relational operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Representation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>not</td>
<td>10</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>3</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>2</td>
</tr>
</tbody>
</table>

Multi-dimensional Operators

In the previous sections only scalar operators have been considered, whereas Booster also supports the use of their multidimensional equivalent. The operators (+, -, *, /, **, div, mod) and (<, <=, =, =>, >=, >) are defined for expressions of any size and dimension. A more formal description of these operators is given in Chapter 7.

Standard Functions

A number of standard functions is available in Booster. These functions can be categorised in two groups: arithmetic and selection functions.
Arithmetic Functions

The intrinsic arithmetic functions are listed in Table VII. In this table, D equals the data type INT or REAL.

Table VII Intrinsic functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Representation</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin, cos, tan</td>
<td>sin, cos, tan</td>
<td>D → REAL</td>
</tr>
<tr>
<td>absolute value</td>
<td>abs</td>
<td>D → D</td>
</tr>
<tr>
<td>is positive</td>
<td>pos</td>
<td>D → BOOL</td>
</tr>
<tr>
<td>square root</td>
<td>sqrt</td>
<td>D → REAL</td>
</tr>
<tr>
<td>exponentiation (e)</td>
<td>exp</td>
<td>D × D → REAL</td>
</tr>
<tr>
<td>random function</td>
<td>random</td>
<td>INT → REAL</td>
</tr>
<tr>
<td>natural logarithm</td>
<td>ln</td>
<td>D → REAL</td>
</tr>
<tr>
<td>round function</td>
<td>round</td>
<td>D → INT</td>
</tr>
</tbody>
</table>

These functions are also applied element-wise: the application of any of these functions on a structure (shape or viewed shape), yields a structure with the specified function applied to each of the elements. For example:

```
SHAPE A, B (n) OF REAL;
A := sin(abs(A));
B := ln(A) * exp(B);
```

Selection Functions

Booster features two selections functions: the minimum and maximum function, which are represented as min and max, respectively. The function max, if applied to a shape or a content expression, returns the maximal value of that shape or expression. The min returns the minimal value of that shape or expression. Both functions can also be applied in the index domain. In that case they return the corresponding index of the maximal and minimal element and are denoted as minindex and maxindex.

These selection functions are frequently used in algorithms with data dependent views. The following example describes the selection of a maximal element, or elements, of a certain shape:

```
SHAPE A (n) OF REAL;
M <- A[maxindex(A)];          (or alternatively M <- A[maxindex];)
R <- A\ maxindex(A)];        (or alternatively M <- A\ maxindex;)
```

The above two view statements define a view identifier M, which refers to the maximal element of A. The view identifier R refers to all, but the maximal element. When these functions are applied to multi-dimensional structures they return an index vector.
Dimension Conversions

As the language allows for the definition and manipulation of multi-dimensional structures, it is necessary to have conventions on implicit or explicit dimension conversions in assignments if dimensions of the left and right part do not match. The following conventions regarding dimension conversions are used:

- Shapes and view identifiers of the same dimension, size, and cardinality may always be assigned to each other;
- Constants and scalars, i.e. one-dimensional shapes, may always be assigned to a (viewed) shape of arbitrary dimension;
- All other assignments must be defined explicitly, that is matching dimensions must be created with help of views.

Reduction Function

The Booster language has a function that can be used to describe repeated function application. This is a functional element in the language which makes it possible to describe more complex programs in the form of expressions. This function, the reduction takes two arguments. The first argument is a dyadic function or operator and the second a content expression of a data-type for which the function or operator is defined. The result of applying the reduction function is the application of the dyadic function or operator on every element of the structure that is returned by the expression. The format of the reduction function is:

\[ \text{REDUCE}(f, \text{expression}) \]

The function or operator \( f \) used for the reduction-operator is assumed to be associative and commutative, similar to addition and multiplication. An example of applying the reduction-function is the following:

\begin{verbatim}
SHAPE V (4) OF REAL;
REDUCE (+, V);
\end{verbatim}

The above REDUCE operator returns the addition \( V[0] + V[1] + V[2] + V[3] \). The product of all elements of a matrix can be described as follows:

\begin{verbatim}
SHAPE M (2 # 2) OF REAL;
REDUCE (*, M);
\end{verbatim}

This yields the product of all the elements in the matrix \( M[0,0] \times M[0,1] \times M[1,0] \times M[1,1] \). The default execution order for the reduce is left to right.
4.4.3 Functional Relations in Content Statements

In Section 4.3 the definition of functional relations on views by using free variables was introduced. This same construct can also be applied to content statements. In its most simple form they can be used to describe straightforward assignments, where A and B are shapes with an index set \( \{0,\ldots,n-1\} \):

\[
\begin{align*}
A[i] &:= B[i]; \\
A[i] &:= i;
\end{align*}
\]

In both content statements the free variable \( i \) is bound to the range \( \{0,\ldots,n-1\} \), resulting in the first statement in a straightforward element-wise assignment and in the second statement in an initialization of the shape-elements with the corresponding index value. Free variables in content statements must occur free somewhere in the content statement. For example,

\[
\begin{align*}
\text{SHAPE C (2*n # m) OF INT;} \\
C[2*i, j] &:= A[i]*B[j \mod n];
\end{align*}
\]

The free variables can also be used in conjunction with simple views, e.g.

\[
A[2:3][i] := 2*i;
\]

In this statement the composition of the selection view \( [2:3] \) with the view \( [i] \) results in the binding of free variable \( i \) to the indices \( \{2,3\} \). The effect of this statement is equal to that of the following two statements:

\[
\begin{align*}
A[2] &:= 4; \\
\end{align*}
\]

Note that in contrast to view statements, there is no renumbering of indices after a simple view assignment. For example, the combination of the following view- and content statements has a different effect:

\[
\begin{align*}
V &<- A[2:3]; \\
V[i] &:= 2*i;
\end{align*}
\]

After the view statement, the view identifier \( V \) refers to the second and third element of \( A \) and \( V \) has an index set \( \{0,1\} \) associated with it. When the simple view \( [i] \) is applied to \( V \) the free variable binds to the index set of \( V \) and hence the effect of the statement (on \( A \)) is:

\[
\begin{align*}
V[0] := 0 &\quad \Rightarrow \quad A[2] := 0; \\
\end{align*}
\]
Non-rectangular Views in Content Statements

In Section 4.3.1, it was stipulated that free variables may occur in integer- as well as index expressions. Up to now, however, all examples involved integer expressions, without considering index expressions. Index expressions can be very useful to define non-rectangular views. For example, consider the selection of the lower triangle of a matrix:

```
SHAPE A (n # n) OF INT;
A[i,0:i] := A[i,0:i]*2;
```

The free variable $i$ is bound to the index set $0:n-1$, where for a given $i$ the corresponding rows $0:i$ are selected. Again, the cardinality of the arguments on the left and right side of the `:=` symbol must be equivalent for each instance of $i$.

![Figure 9 The selection of the lower triangle of a matrix](image)

Functional Relations with Dependencies

Functional relations in content statements presented so far, are all element-wise operations. For example:

```
SHAPE V,W (n) OF INT;
V[i] := i;
W[1:upb][i] := V[i-1]+1;
```

For every $i$ in $1:n$, the calculation $W[i] := V[i-1] + 1$ is independent of any other computation of an element of $W$: in other words there is no data dependence. Consider the situation where an intra-statement data dependence is introduced by replacing $W$ by $V$.

```
SHAPE V (n) OF INT;
V[0] := 0;
V[1:upb][i] := V[i-1]+1;
```

In the case of an intra-statement dependence, the computation of $V$ is not defined element-wise, but is sequentialized. That is for the computation of $V[1]$, $V[0]$ is needed, for $V[2]$, $V[1]$ is needed, etc.

The situation is even more complicated if multiple dependencies are used, consider:
SHAPE V (n) OF INT;
V[0] := 1;
V[n-1] := 2;
V[1:upb-1][i] := V[i-1] + V[i+1] + 1;

The last content statement does not only introduce a double data dependence, but is also ambiguous, since two sequentializations are possible: the calculation of \( V[1] \), followed by \( V[2] \), etc., or the calculation of \( V[n-2] \), followed by \( V[n-3] \), etc. This semantic ambiguity is overcome in the language by introducing the following sequentialization convention:

**Sequentialization Convention**

If a content statement in *Booster* has more than one intra-statement dependence, than of all possible evaluation orders the one is selected that starts with an element having an index with the smallest ordinal. This ordinal is determined with the *ord* function, which returns a unique number for each index, as was discussed in Section 3.3.2. This ordering corresponds with the lexicographic ordering of index sets.

The rationale for choosing this convention is that if a programmer specifies a relation that contains an ambiguity, it is apparent that a certain relation must hold for every element of that particular shape. In such a case, a sequentialization convention as described above, forces the evaluation order to be deterministic. From this convention, every required sequential order can be realised by using the appropriate views.

If we return to the last example, our convention has as a consequence that the evaluation order \( V[1], V[2], \) etc. is selected instead of \( V[n-2], V[n-3], \) etc. Another example, which is slightly more complicated, is a multidimensional case. Consider the following example:

```plaintext
SHAPE A (n # n) OF REAL;
A[_,0 | n-1] := 1.0;    ( Initialize the boundaries of A. )
A[0 | n-1,_] := 1.0;
```

In this particular example, there are four possible evaluation orders ("the four corners of the matrix"). The evaluation order that starts with element \( A[1,1] \) is selected, since the index \((1,1)\) comes first in the ordering. This computation and its default evaluation order is depicted in Figure 10.
4.4 CONTENT OPERATIONS

4.4.4 Data Sets

The concept of data and data-sets has been discussed in close connection with shapes. Although examples of scalar values have been introduced, the concept of separate data sets has not been considered. We distinguish two kinds of data sets: shape constants and data fields. A shape constant is an ordered set of data values, and a data field is a functional description of a data set, which is instantiated when used.

Shape Constant

A shape constant is an ordered set of integers, reals, or Booleans which are delimited by curly brackets. For example, consider the following constants:

```plaintext
SHAPE A, B (2 # 3) OF INT;
A := {{1,2,3},{4,5,6}};
B := A + {{1,1,1},{1,1,1}};
```

Here two shapes with 2 rows and 3 columns are declared. The shape is initialized to an equally dimensionalized compound constant. After evaluation of both statements the value of B is equal to \{\{2,3,4\}, \{5,6,7\}\}.

Data Fields

A data field is described by a view statement. Instead of a single shape, a view statement defining a data field may have any content expression on the right side of the '<-' symbol. An example is a definition of squares:

```plaintext
SHAPE A (10) OF REAL;
V (20) [i] <- i*i;
A := V[0:9]*2;
```

The view statement defines a data field \{0,1,2^2,3^2,4^2,\ldots,19^2\} which has an explicit index set \{0:19\}. This data field is used in the content statement in conjunction with a simple view function \{0:9\}. This means that not the whole range of V is used, but only a subset. Eventually, the set of values \{0,2*1,2*2^2,2*3^2,\ldots,2*9^2\} is
assigned to a. Another example of a data field is to use expressions with shapes and operators in a view statement:

\[
\text{SHAPE A, B, C (n) OF REAL;}
\]
\[
V \leftarrow A \times B / C;
\]

An obvious restriction is that view identifiers defining a data field cannot be used on the left side of content statements. The rationale for these kind of data fields is the possibility to identify certain subexpressions without the necessity to introduce new shapes and therefore storage.

4.4.5 Functions

*Booster* supports the encapsulation of content-, view-, and control statements in *functions* that are interfaced via a formal argument list. The formal arguments of functions are, like those of view functions, divided into input and output arguments, where the input arguments are to the left of the '→' symbol and the output to the right. The arguments are either *index*- or *shape* arguments, where the use of index arguments is restricted to the input. As in view functions, the cardinality of the shape arguments has to be declared, and in addition the data type of shape arguments must be specified. In order to implement a meaningful function, it is necessary to have at least one input and one output argument.

Moreover, the function declaration allows for the — optional — declaration of (local) shapes in functions, whose scope is restricted to the function body only. Neither these local shapes, nor any of the locally defined view identifiers, are retained between invocations of the function. Furthermore, functions and view functions in *Booster* are free of side effects: apart from the formal arguments, local shapes, and locally defined view identifiers no other structures may be used. Finally, in contrast to the view functions, recursion for *Booster* (content) functions is not permitted. The rationale for this restriction is discussed in Chapter 6 and onwards. The template for a function declaration is:

\[
\text{FUNCTION function-id [ PRIORITY integer-expression ]}
\]
\[
(\arg{1}, \arg{2}) \rightarrow (\arg{3}, \arg{4});
\]
\[
\{ \text{arg-def; } \}^*\]
\[
[ \text{SHAPE shape-decl; } ]\]
\[
\text{BEGIN}
\]
\[
(\text{statement; })^*\]
\[
\text{END;}
\]

The function declaration also allows for the specification of a priority of a function. This priority is needed to derive the evaluation order, if a function is used in *infix* instead of *prefix* notation. In other words, functions may be used as op-
operators in the program as long as their priority is defined and a function has at most one output shape argument and one or two input shape arguments.

Below, we illustrate the function declaration by implementing the inproduct for two vectors:

```
FUNCTION InProd ( A, B ) -> ( InProduct );
A (n) OF REAL, B (n) OF REAL, InProduct (1) OF REAL;
BEGIN
  InProduct := REDUCE(+, A*B);
END;

SHAPE V, W (20) OF REAL;
  Z (1) OF REAL;
Z := InProd(V,W);
```

The function InProd is first declared with two input shape arguments and one output shape argument. Both cardinality and data type of each of the shape input arguments are declared. The body of the function consists of one content statement, where the inproduct is described by means of a reduce operator. In the content statement below, the function is used with actual arguments V and W. As was stipulated above, functions can also be used in infix notation under certain conditions. For example, the last line of the above example could be replaced by:

```
Z := V InProd W;
```

When a function is used in infix notation however, an operator priority has to be defined for it. This is done in much the same way as for standard operators, namely by assigning a number between 1 and 10 to the function. We have to change the function declaration of InProd accordingly by assigning the priority value 7:

```
FUNCTION InProd PRIORITY 7 (A, B) -> (InProduct);
```

... ...

A more complex example is the function matrix_product.

```
FUNCTION matrix_product PRIORITY 7 (A, B) -> (C);
A (n ≠ m) OF REAL, B (m ≠ k) OF REAL, C (n ≠ k) OF REAL;
BEGIN
  C[i,j] := REDUCE(+, A[i,_] * B[_ ,j]);
END
```

---

4 Note that for vector inproduct the same priority is chosen as for multiplication and division.
This function matrix_product has a $n \times m$ matrix and $m \times k$ matrix as input and returns a $n \times k$ matrix. In the function body the operational definition of matrix_product is given.

4.5 Control Statements

The flow of control of Booster programs is governed by two statements: the repetition-, and if-then-else statement. The repetition-statement defines a repetition of its body either a fixed number of times or dependent on some condition. The if-then-else-statement defines conditional execution. A last "control" statement in Booster is the index statement. This statement implements a restricted macro expansion and does not add anything to the semantics of the language. This macro expansion can only be applied to expressions used in the index domain. The rationale is that these expressions sometimes tend to become rather large and expression are often used more than once in the program.

4.5.1 Repetition Statement

The Booster language knows a single construct to define repetitions, the repetition-statement. This repetition-statement consists of a so-called while part and iter part. The while part is typically used to specify repetitions that terminate on some condition which might be dependent on data produced in the loop-body. The iter part, however, is used to specify a number of iterations that is independent of the data produced in the loop-body. Although the iter part can be expressed by a while statement, the distinction has been made to allow for a more efficient translation.

While-Statement

Consider the following example declarations:

```
SHAPE S (10 # 10) OF INT;
SHAPE X (10) OF INT;
```

A typical example of a while statement, where data produced in the statement body is used in the condition of the loop, is:

```
X := .;
S := .;
WHILE S InProd X > 0 DO
  S[i,j] := S[i-1,j] + S[i+1,j] + S[i,j+1] + S[i,j-1];
END;
```

In the above example, the function InProd, which yields the inproduct of two vectors, is used in infix notation. As an example of an iter-loop, where no dependence exists between data in the loop-body and the iteration parameter $n$, consider:
\[ S := \ldots; \]
\begin{verbatim}
ITER n DO
    S[i,j] := S[i-1,j] + S[i+1,j] + S[i,j+1] + S[i,j-1];
END;
\end{verbatim}

The main reason for having two repetition-constructs is translation efficiency. Not many optimisations are possible in the translation of a while-part: only one iteration can be executed at a time, followed by the evaluation of the loop-condition. For the iter-part, however, iterations can be executed in parallel, provided that the data dependencies are respected, since the number of iterations is always known in advance. This in contrast to the while-part, where every iteration could be the last one and multiple iterations on the same data structure would result in incorrect data. The Gauss-Seidel algorithm, which is partly specified above, is a perfect illustration of the use of the iter-part.

The general repetition-statement

The above language constructs are both special cases of the following general repetition statement:

\[ \text{repeat-statement} ::= \text{repeat-specification DO } \]
\[ \{ \text{statement } ; \}^+ \]
\[ \text{END} \]

\[ \text{repeat-specification} ::= \]
\[ \text{ITER integer-expr | WHILE conditional-expr | ITER integer-expr WHILE conditional-expr} \]

The statements in the body of this construct are repeated as many times as specified following the iter-keyword as long as the conditional expression of the while is true.

4.5.2 Conditional Statement

For conditional execution the if-then-else statement is available. The template for this statement is:

\[ \text{IF conditional-expr THEN } \{ \text{statement } ; \}^+ \]
\[ \text{ELSE } \{ \text{statement } ; \}^+ \] \]
\[ \text{END} \]

The conditional statement executes the statements in its then- or else-part depending on the result of the evaluation of the conditional-expression.
4.5.3 Index Statement

As was mentioned above, the index statement implements a restricted kind of macro expansion for the index domain. Index statements come in two flavours: the local, non-parameterized index statement and the global, non-parameterized statement. The prime use of the first one is to save effort by offering the option to substitute a single expression on more than one place in the program. The second parameterized version can be used to specify general expression-templates which can in turn be used on several places in the program. The syntax-template of this statement is as follows:

\[
\text{index-statement} ::= \text{INDEX} \text{ index-variable [(arg-list)]} = \text{index-expr} \\
\quad \{, \text{index-variable [(arg-list)]} = \text{index-expr} \}
\]

The use of this construct is illustrated in the examples given in Section 4.6.2:

4.6 Modules

4.6.1 Module Template

The standard encapsulation facility in Booster is the module. A module has an interface definition like a function. The main difference between a module and a function is however, that modules are the interface to the outer world. The integration of Booster programs with programs in other formalisms is always handled via the module. Furthermore, annotation modules are always associated with modules, whereas functions are merely used to encapsulate functionality and to realise standard software engineering principles of maintainable code and software layering. The programmer can build its own software interface by designing and implementing a set of (view) functions.

Constant definitions

Modules also allow for the definition of so-called constants. These constants can be used in the module in the data domain. Once a constant is defined in the constant definition section it may never be changed. The template of constant definition is:

\[
\text{CONST } \text{constant-id} = \text{constant-expr} ; \\
\quad \{ \text{constant-id} = \text{constant-expr} ; \}
\]

An example of a constant definition is:

```
CONST pi = 3.1415;
c = 4;
twopi = 2*pi;
```

The template of the module is specified below:
4.6 MODULES AND EXAMPLES

MODULE moduleId [( argIdList )] -> ( argIdList );
shapeDeclList;
[ CONST constDeclList ;]
[ viewFunctionDeclList ;]
[ functionDeclList ;]
[ SHAPE shapeDeclList ;]
BEGIN
  [ statementList ;]
END.

An example of a module specification is the following:

MODULE Calculate_trace ( A, B ) -> ( Trace );
A, B (n # m) OF REAL; Trace (1) OF REAL;

VIEW FUNCTION diagonal ( X ) -> ( Y );
X (n # n), Y (n);
BEGIN
  Y[i] <- X[i,i];
END;

FUNCTION matrix_mult ( X, Y ) -> ( Z );
X, Y, Z (n # n);
BEGIN
  Z[i,j] := REDUCE(+,X[i,_] * Y[_ ,j]);
END;

BEGIN
  Trace := REDUCE(*,diagonal(matrix_mult(A,B)));
END.

This module computes the trace of a matrix that results from a matrix-multiplication. To this purpose, a view function and a function are defined. The view function diagonal selects the diagonal of a square matrix and the function matrix_mult implements the matrix multiplication. These two functions form the building blocks for the formulation of the trace-computation. The computation is straightforward: after the matrix product of shapes A and B, the diagonal of the result is taken and is reduced with the multiplication operator. It should be noted that the actual computation is optimised and does not involve computation of the matrix product, but only the necessary part.

4.6.2 Examples

In this subsection a number of example algorithms is presented, each of which illustrate various aspects of the Booster language. A number of the examples will be used in Chapter 6 to illustrate the translation of Booster.
A. LU-Factorization

LU-factorization is a frequently used algorithm as a first step in the solution of a linear set of equations. This algorithm illustrates the essence of Booster programming: start with the identification of the basic data structures or shapes. Next, identify the different regions of these shapes which are treated as a coherent set within the algorithm. These regions must also be considered in the evolution of the algorithm and in relation to each other. The following step is to formalise the regions in terms of Booster's view concept. Finally, the algorithm is described in terms of these views.

We will now apply this general scheme to our example. LU-factorization only comprises one input data structure, a (square) matrix, and yields two triangular matrices which, when multiplied, yield the input matrix. The algorithm consists of a number of distinct phases or iterations, where in each iteration four regions are of interest: the pivot-element, the pivot-row, the pivot-column, and the remainder, which are referred to as P, R, C, and B, respectively. The relations between these — non-overlapping — regions are depicted in Figure 11.

![Figure 11](image)

*Figure 11 Representation of the regions involved in the algorithm.*

These regions are extracted from the original input data structure by using partial pivoting on the first column of the matrix. After this initial step, every next set of regions P, C, R, and B is derived by considering the previous region B — the remainder — as the new matrix to be acted upon. This process is depicted in Figure 12.

![Figure 12](image)

*Figure 12 Relation between the old regions — P', R', C', and B' — and the new — P, R, C, and B — in the evolution of the algorithm.*
The next step is to formalise the regions $P, C, R,$ and $B$ in *Booster* views. We observe that the definitions of these regions are very closely related, because from the selection of the pivot element every region is derived. In addition, the derivation of the regions is repeated every iteration, which makes it evident to encapsulate their definitions in a (single) view function. Prior to the view function definition, we have to realise what the cardinalities of the different regions are and how they are related to each other. In this case, the pivot element is always a single element, both the pivot-row and column are vectors with a length of one smaller than the size of each side of the matrix. Finally, the remainder is also a square matrix with sides identical to the size of pivot-row and column. The formulation in *Booster* is straightforward:

```
VIEW FUNCTION Pivoting (Q) -> (P, R, C, B);
Q (m # m); P(l); R, C (m-l); B ( (m-1) # (m-1) );
BEGIN
  INDEX pivot = maxindex(abs(Q[_,0]));
  P <- Q[pivot, 0];
  R <- Q[pivot, 1:upb];
  C <- Q[\pivot, 0];
  B <- Q[\pivot, 1:upb];
END;
```

In the argument declaration list, the aforementioned relations between the regions are expressed. The body of the view function implements the pivoting operation with the help of the intrinsic functions *maxindex* and *abs*. This example also illustrates the use of the index statement. The value *pivot* is needed for each of the four view statements and is therefore defined by using the index macro facility.

Having defined all regions, we can formulate the algorithm:

```
(P, R, C, B) <- Pivoting(A);
WHILE SIZE(B) > 1 DO
  C := -C/P;
  B[i,j] := C[i]*R[j] + B[i,j];
  (P, R, C, B) <- Pivoting(B);
END;
```

Preceding the repetition-statement, the input matrix $A$ is viewed in four desired regions. The repetition is executed as long as the size of the remainder $B$ is larger than 1. The body of the repetition contains two content statements that implement the factorization operation. The first one normalises the pivot-column, the second one updates the remainder $B$. Note that removal of the first
content statement of the `while`-part of the repetition-statement — \( C := C/P; \) — changes the LU-factorization to a Gaussian elimination.

**Forward and Backward substitution**

Once the matrix is factorised in a \( L \) and \( U \) lower and upper triangle, the set linear equations can be solved as follows:

\[
LUx = b \quad \Rightarrow \quad Lc = b \quad \text{forward substitution}
\]

\[
c = Ux \quad \text{backward substitution}
\]

In *Booster* this is expressed as follows:

\[
\text{SHAPE C (n) OF REAL;}
\]

\[
C[i] := B[i] - A[i, 0:i-1]*C[0:i-1];
\]

\[
X[i] := (C[i] - A[i, i+1:upb]*X[i+1:upb])/A[i, i];
\]

**B. Successive Over Relaxation (SOR)**

The next example, Successive Over Relaxation (SOR) [Adams83], uses another technique common in the *Booster* language: dimension increasing views. SOR is a technique to solve partial differential equations, such as Laplace's equation. In SOR, each point on a grid is updated by the mean value of its nearest neighbours, plus a correction factor \( c \). In *Booster* this is formalised in the function SOR:

\[
\text{MODULE SOR (A) \rightarrow (A);}
\]

\[
A \ (n \ # \ n) \ \text{OF REAL;}
\]

\[
\text{FUNCTION SOR (X, c) \rightarrow (Y);}
\]

\[
X \ (n \ # \ n); \ Y \ (n-2 \ # \ n-2);
\]

\[
\begin{align*}
Y[i, j] := & \ (X[i, j+1] + X[i+2, j+1] + X[i+1, j] + \\
& X[i+1, j+2])/(c/4) + (1-c)*X[i+1, j+1];
\end{align*}
\]

\[
\text{BEGIN}
\]

\[
\text{VIEW FUNCTION interior (X) \rightarrow (Y);}
\]

\[
X \ (n \ # \ n); \ Y \ (n-2 \ # \ n-2);
\]

\[
\begin{align*}
Y & \leftarrow X[1:upb-1, 1:upb-1];
\end{align*}
\]

\[
\text{BEGIN}
\]

\[
\text{ITER k DO}
\]

\[
A[\text{interior}] := \text{SOR(A, c)};
\]

\[
\text{END;}
\]

\[
\text{END.}
\]

Here, \( c \) is a real number ranging from 0 to 1 and \( k \) is a integer constant indicating the number of iterations. Dependence analysis on the functions content statement reveals that ambiguities arises in the execution order. Hence, by default the updates on \( A \) are done in lexicographic order, yielding the required
execution order. SOR can also be defined using the red/black ordering method [Fox88]. This algorithm orders the matrix like a chessboard with blocks of size \( s \) of which half the squares is red and half is black, where the red and black areas are chosen such that the borders overlap each other with single points. Next, the red/black algorithm applies the SOR algorithm to the each of the red blocks first, followed by the application to all black blocks.

The selection of regions of the red/black algorithm is rather complex and we will therefore break up the selection in a number of distinct phases. First, we describe the selection of a single block of size \( s \) from the matrix. To this purpose, we will use a view function Block that takes a square matrix \( X \), a size \( s \), and a position \( v \) as input and yields the corresponding — overlapping — block \( Y \) of size \( s+2 \). The positions of the blocks are numbered from left to right, from top to bottom on the matrix (see Figure 13).

![Figure 13 Numbering of blocks](image)

The following two index statements implement this numbering:

\[
\text{INDEX XPos}(v,n,s) = v \text{ div } ((n-2) \text{ div } 2), \\
\text{YPos}(v,n,s) = v \text{ mod } ((n-2) \text{ div } 2),
\]

Finally, the size of each block is equal to \( s + 2 \), including the outer boundary. The view function Block is defined with help of the index statements:

\[
\text{VIEW FUNCTION Block} (X, \text{ overlap, } s, v) \rightarrow (Y); \\
X (n \# n); Y (s+\text{overlap}+2 \# s+\text{overlap}+2); \\
\text{BEGIN} \\
Y[i,j] \leftarrow X[\text{XPos}(v,n,s)*s+i, \text{YPos}(v,n,s)*s+j]; \\
\text{END};
\]

The above view function is in turn used for the next view function \text{red_black}, which effectively splits the input argument \( w \) in the red and black overlapping regions. We use a parameterized index variable to reduce the size of the index domain definitions of the view function:
INDEX numberR(x, y) = (x-2)*(x-2) div (y*y*2) - 1,
      numberB(x, y) = (x-2)*(x-2) mod (y*y*2) + numberB(x, y);

VIEW FUNCTION red_black (W, s) -> (R, B);
W (n # n);
B (numberB(n, s) # s+2 # s+2);
R (numberR(n, s) # s+2 # s+2);
BEGIN
  B[u, _, _] <- W[Block(1, s, 2*u)];
  R[u, _, _] <- W[Block(1, s, 2*u+1)];
END;

The view function red_black takes a two-dimensional array W as input and
returns two three-dimensional arrays R (ed) and B (lack) as output. In fact,
all Red and Black sub-arrays are stacked in the u-direction, which is depicted
below in Figure 14. (The solid lines indicate the — single element — overlapping
areas.)

![Figure 14 Red/Black ordering](image)

The red/black program is now expressed in terms of the — stacked — view
identifiers R and B and the SOR function is applied to each element.

SHAPE P (n # n) OF REAL;

(B, R) <- red_black(P, size);
ITER k DO
  R[k, interior] := SOR(R[k, _, _], c);
  B[k, interior] := SOR(B[k, _, _], c);
END;

where size is a parameter that defines the sizes of the blocks, k denotes the
number of iterations, and c is the constant used in the SOR algorithm. As was
mentioned before, the algorithm implements Successive Over Relaxation by
first updating all red blocks, followed by all black blocks. In the next chapter, we
will explore the parallel properties of this algorithm.
C. Simplex Method

The simplex method [Dantzig63] is an algorithm that is frequently used to solve linear programming problems. Linear programming models involve optimisation of some objective function or cost function. From the set of possible solutions the solution is selected for which the objective function attains an optimal value. Formally the problem can be expressed as follows:

Maximise $z = Cx$
Subject to: $x \leq 0$ and $Tx \leq B$

where $T$ is the matrix with coefficients, $C$ is the optimisation vector, and $B$ is the constraint vector. Further details can be found in [Dantzig63]. The simplex method can be visualised as follows:

Figure 15 The template of the Simplex method

Below the Booster version of the simplex method is described. The shape-names correspond with the names used in the figure above.

```
MODULE SimplexMethod (T) -> (Z);
T (n # m) OF REAL;   { The simplex table.   }
Z (1) OF REAL;       { Value of the optimisation criterion. }

IMPORT FUNCTION mult FROM Gaussian_elimination;

VIEW FUNCTION Search (T) -> (pivot, pivot_row, pivot_column, rest);
T (n # m); pivot (1); pivot_row (m-1); pivot_column (n-1);
BEGIN

INDEX pivot_c = maxindex(T[upb,1:upb]),
S = T(_,pivot_c] > 0 && T[0:upb-1,0] > 0,
pivot_r = maxindex(T[S,pivot_c]/T[S,0]);
pivot <- T[pivot_r, pivot_c];
pivot_column <- T[\ pivot_r, pivot_c];
pivot_row <- T[pivot_r, \ pivot_c];
rest <- T[\ pivot_r, \ pivot_c];
```

END;
```
BEGIN

    (pivot,pivot_row,pivot_column,rest) <- Search (T);
    { Select the pivot, pivot row, pivot column and remainder. }

WHILE pivot_column > 0 DO

    pivot := 1/pivot;
    { Calculate the pivot element }

    pivot_row := pivot_row*pivot;
    { Update the pivot row. }

    Z := Z + pivot_column*pivot_row;
    { Update the optimisation criterium. }

    rest := rest - mult(pivot_column,pivot_row);
    { Update the rest of the matrix. }

    pivot_column := -pivot_column*pivot;
    { Update the pivot column. }

    (pivot,pivot_row,pivot_column,rest) <- Pivot_Select (T);
    { Select the new pivot, pivot row, pivot column and rest. }

END;

END.

This example illustrates a more advanced use of index statements. The view function Pivot_Select selects the relevant regions as follows: First, an index variable pivot_c equal to the maximum of the constraint vector B is defined. Next, an index variable S is defined that contains the indices of the elements for which both the values of the constraint vector B is larger than zero, as well as the values of the column of T — for which the optimisation vector C is maximal — are larger than zero. Finally, both index variables are used for the definition of pivot_r that equals the maximal index of the quotient of rows S of T and B. Having defined the index variables, the selection of the appropriate regions pivot, pivot_column, pivot_row, and rest is straightforward. These regions in turn enable the transient specification of the simplex algorithm.
Chapter 5
Exploiting the Parallel Virtual Machine

An algorithm described in Booster does not presuppose any property of the machine it is eventually executed on. Evidently, this machine-independent description permits the translation of Booster programs to a variety of (parallel) machines. Ideally, the translation process is automatic, but unfortunately state-of-the-art compilers are not capable of producing efficiently executing code for arbitrary algorithm-machine combinations. One of the reasons for this inability is the lack of additional algorithmic or machine-specific information. For example, it is not clear from a given algorithm described in Booster whether or not it is safe from the viewpoint of numerical stability to interchange operations. This meta-information has to be supplied explicitly by the programmer to allow performance enhancing transformations. Furthermore, additional information is needed to control the complexity of the translation. The translation of a Booster program without any additional information might involve the internal generation and evaluation of many thousands of different parallel programs. For the evaluation of these parallel programs one needs a performance model to predict performance of a certain program on an arbitrary parallel machine. This is still an open problem. The compiler therefore needs information on potentially successful partitionings or decompositions of the program. A last argument in favour of explicit additional information is the fact that this allows for experimentation with different algorithms and partitionings. Various alternative algorithms can be compared and new, perhaps better, algorithms can be developed in this way. Here, the compiler is only used as a means to facilitate experimentation, while leaving the essential elements of the translation under control of the programmer.

Information about the nature of the input data of a certain algorithm can also be of great importance to the translation of algorithms into efficiently executing parallel programs. This is especially true for those algorithms that involve sparse data structures. The storage of sparse data structures as dense ones can lead to severe performance degradation. This can be overcome by informing the compiler about which value to store and which not.¹

¹ Note that in imperative languages sparsity information is often hard-wired into the algorithm by the use of index arrays.
One must keep in mind that the above additional information is not essential to the algorithm and does not change the semantics of the algorithm in terms of the input-output behaviour. The mere purpose of the additional information is to tune and optimise the generated code for a certain class of parallel machines. This offers the programmer the opportunity to experiment incrementally with different mappings of computations and storage patterns, until a satisfactory solution has been reached. In other words, a programmer can specify additional information on any desired level of detail, effectively finding a trade-off between time invested in tuning and optimisation of the translation and speed-up of the resulting program gained.

The language concepts for this annotation language have been chosen in Chapter 3. Amongst the language concepts are the virtual machine and the definition of a mapping of the program to that machine. The resulting parallel program for the virtual machine can in turn be mapped on the real machine. This two-phase mapping enables the "re-use" of program annotations, because if a real machine with for example more processors is used we only need to change the mapping from virtual machine to real machine.

The discussion on the annotation language is organised as follows. In Section 5.1, the virtual and real machine and their basic building blocks, processors and memories, are discussed. Section 5.2 discusses the syntax of the annotation language and the template for specifying relations between programs and machines. In Section 5.3, available mappings between the data structures in the algorithm and the elements of the virtual machine capable of storing data are discussed. In Section 5.4, the actual introduction of parallelism takes place: the computational organisation of the algorithm by assigning responsibility for computation to processors. In the fifth and last section, the implications on parallel algorithm design of our approach are explained by relating them to a number of parallel programming techniques. These techniques and their implications are explained by annotating the examples discussed in Section 4.6.2.

5.1 Relating Booster, Virtual Machines, and Real Machines

As presented in Section 2.1, we distinguish three levels of abstraction in the process of making (parallel) programs. The first level is the Booster language itself, where shapes and views are the programmer's conception of the organisation and interrelation of data. The second level of abstraction is the virtual machine level. The virtual machine reveals the notion of parallelism and data organisation to the programmer. By relating data and computation to the entities processors and memories of the virtual machine, additional information on the locality of computation and data of the program can be expressed. Here, the
5.1 RELATING BOOSTER AND MACHINES

Virtual machine may have more processors and memories than are available on the target parallel machine. The third level of abstraction is the real machine, that models the actual target parallel machine. The real machine, in contrast to the virtual machine, also has the notion of an interconnection topology.

The motivation for expressing parallelism at three levels of abstraction will be made clear in the following. Information to tune the translation of a Booster program is formulated in terms of data- and computation organisation. Shapes defined in a Booster program are the programmer's conception of the organisation of the data within the algorithm. However, from the viewpoint of the virtual machine, shapes are merely considered as views on its memories. By relating the actual representation on a virtual machine and the shape in question through a view, the programmer can alter the place and ordering of data in the evolution of the algorithm. In the same way, the programmer can define the processors responsible for the computation of certain data. This organisation of the computation determines the (parallel) execution behaviour of the parallel algorithm. The relationships between shapes, views, and virtual machines are depicted in Figure 16.

![Diagram](Image)

**Figure 16. The concept of data and computation organisation.**

Views on shapes, as well as the organisation of the data on the underlying virtual machine, are described by the same formalism. The view concept of Booster with the corresponding syntax can be used directly to describe the relationship between shape and virtual machine. In turn, this virtual machine is described in terms of a real machine model. This model basically has the same building blocks as the virtual machine: processors and memories, and imports the concept of the interconnection structure but this is not visible to the programmer and is only used by the compiler. This is depicted in Figure 17.
In the next subsections, we will introduce the building blocks, processors and memories for virtual machines and in addition interconnection structures for real machines.

5.1.1 Processors

One of the building blocks of virtual and real machines is the processor. A processor — in this context — is a device that is capable of processing a sequence of instructions in a single address space. We assume that all processors are identical on the virtual machine-level, although this is not necessary in the real machine. The processors of a virtual machine are ordered in a so-called processor structure, where each processor is assigned a number. In the annotation language, it is possible to declare these processor structures in the same way as shapes are declared in Booster. The processor structure declaration is preceded by the keyword PROC. For example:

```
PROC Linear (p);
PROC Mesh (p ≠ q);
```

The first declaration defines a virtual machine Linear with p processors numbered linearly. The second declaration defines a virtual machine Mesh with p×q processors numbered in two dimensions. In this way, virtual machines of arbitrary size and — rectangular — structure can be declared.

5.1.2 Memories

A memory is a device that can store data and instructions. On the level of the virtual machine all memories, which may vary from background memory such as tape-units and disks to fast cache memory, are treated alike. The rationale for this is the fact that we want portability of virtual machines across a number of real machine which may have similar interconnection topologies and an equal number of processors, but varying sizes of core and background memory.

The only distinction between memories made on the level of the virtual machine is that of locally versus globally accessible memory. More general; there are memories which can be accessed by a certain processor without intervention of any other processor and there are memories for which at least
one different processor is needed to access data residing in those particular memories.

On the level of the real machine, we make a distinction between those memories that can be directly accessed by at least one processor without the need to pass through other — intermediate — memories and those that can not be directly accessed. Examples of these are the main-memory of a computer system, which is accessible to at least one processor, and disk-memory, which can only be accessed through intermediate memory.

Definition of memories

A virtual machine may have one or more distinct memories, that is one or more memories that do not have the same address-space. Memories are declared like shapes, but if there is more than one distinct memory, an additional declaration of the local structure of each memory is required. The memory of the system is represented as a multi-dimensional structure, where the first cardinality declaration represents the numbering of the memory-structure and the second cardinality declaration the ordering within each memory. Some examples:

\[
\text{MEM X (n)}; \\
\text{MEM Y (n) (m)};
\]

The first declaration defines a memory \( X \) with \( n \) elements, whereas the second declaration defines a group \( Y \) of \( n \) memories with \( m \) elements each.

Processors obtain the data to be processed from memories and store the results again in memory. At first sight, this model restricts the target machines to von Neumann type of architectures. This is not true, because for example the buffers of systolic and array processors can be considered as memories too. As a consequence, modelling of and mapping to these kind of machines is also feasible, which is demonstrated in Section 5.5.2.

5.2 Annotation Language

All definitions regarding virtual and real machines and the various mappings on them are assembled in a so-called annotation module. Each Booster program may have several associated annotation modules, containing mapping information for a specific (class of) parallel machines. The reason for separating the annotation information from the algorithm is that this enables the replacement of annotations for a certain machine by those for a different
machine without altering the actual algorithm. The principle of separate annotation modules is depicted below in Figure 18.

![Diagram](image)

**Figure 18. The relation between Booster and Annotation modules.**

To put the annotation language into context, we return for a moment to the discussion on the evolution from *problem statement* through *algorithm approach* to *virtual algorithm* and *program* in Section 1.3. Here, the *Booster* language is used to describe algorithms on the *virtual algorithm* level. The mapping in the annotation language to a virtual machine results in an algorithm on the *ideal algorithm* level. Finally, machine specifics are introduced in the mapping of the virtual machine to a real machine. This reflects the relation between the ideal algorithm level and the (machine-dependent) program level.

The *Booster* language and its annotations only support those algorithm-machine combinations that have the same virtual algorithm. For some machines, a different algorithm approach may have to be selected, resulting in a different virtual algorithm.

### 5.2.1 Annotation Language and its Environment

The annotation module is a coherent set of mapping descriptions of shapes and views to a virtual machine and of the virtual machine to a real machine. Furthermore, each annotation module defines an interface to a specific *Booster* program module and a processor model as defined externally.

The template of the annotation module reflects the interfaces and mappings that were mentioned in the introduction.

---

2 This separation is conceptual in nature: with the appropriate tools, annotations can be used in the same single text as the *Booster* algorithm. The strict separation serves to illustrate the fact that no mutual, hidden dependencies exist.
5.2 ANNOTATION LANGUAGE

ANNOTATION MODULE module-name ;
    virtual-machine-definition ;
    [ view-function-definitions ; ]
    program-mapping ;
    [ machine-mapping ; ]
END.

Each annotation module has its own identifier and one or more virtual machine definitions. Much like Booster, the annotation language — optionally — allows for the definition of view functions. In the program- and machine-mapping sections, the interface is defined to the Booster-module and real machine, respectively. The latter one is optional, the first however, obligatory. In the next subsections, we discuss each part of the annotation module.

5.2.3 Declaration of Virtual Machine(s)

The virtual machine definition gives the programmer the opportunity to create a model of the architecture onto which a certain program is to be mapped. This model can be defined on many levels of abstraction: with or without the notion of memory, with distributed- or shared memory, or with a bounded- or unbounded number of processors. This model creates the possibility to experiment with gradually more complex virtual machines. By adding extra elements to the model, it is possible to isolate effects of certain mappings.

The template of a virtual machine description is as follows:

VIRTUAL MACHINE name ( [ processor-def ] { memory-def } );

The virtual machine definition consists of the elements that were introduced in Section 5.1. Note that the virtual machine definition only allows for the declaration of a single processor space, but allows for multiple memory definitions. The definition of each of these elements is as follows. The processor definition equals

PROC proc-name ( cardinality-spec );

MEM mem-name [ ( cardinality-spec ) ] ( cardinality-spec );

If in the declaration of a memory, only one cardinality specification is given, this defines one single memory with a certain size and structure.

Those virtual machines that have a single memory local to the processor, can be declared in a combined form, by preceding them with the keyword PROCMEM:

PROCMEM proc-mem-name ( cardinality-spec ) ( cardinality-spec );
**Examples**
A shared memory virtual machine vm would be declared as:

```
VIRTUAL MACHINE shared (PROC processor (p); MEM memory (n));
```

Here shared has \( p \) processors and a single shared memory of size \( n \). A distributed memory virtual machine would be declared as:

```
VIRTUAL MACHINE dm (PROC processor (p); MEM memory (n) (m));
```

Here dm has \( p \) processors and \( n \) memories of size \( m \). One could also "model" secondary memory by providing an additional memory specification:

```
VIRTUAL MACHINE diskfarm ( PROC processor (p); MEM memory (n) (m) ; MEM disk (p) (k));
```

This virtual machine diskfarm has, compared to distributed an additional farm of \( p \) disks which each contain at most \( k \) elements.

### 5.2.4 Mapping and Interface to a Booster module

The interfacing to the *Booster* module can be done in two ways: either the entire name-space of a *Booster* program or only a limited number of identifiers can be included. In the first case, no explicit argument has to be supplied. In the second case, the identifiers involved have to be specified. The template for the interfacing and mapping is defined as follows:

```
IMPORT Booster-module-name [ :: arg-list ];

mapping-statements
```

The argument-list in this template declares the identifiers and their cardinality specification, optionally with scoping information, if the identifiers are declared in the scope of functions. The mapping statements can be defined on the virtual machine itself or on its elements: processors and memories. Consider the following example mapping, where the mapping on the processor and memory elements of the virtual machine have been separated. We will elaborate on the mappings themselves in more detail in the next sections.

```
VIRTUAL MACHINE vm ( PROC processor (p); MEM memory (n) );
IMPORT Gaussian-elimination :: A (m # m);
    Matrix-Mult :: X (k # k);

X[i,j] <- memory[i + j*k];
A[i,j] <- memory[i*m + j];
```

The import statement specifies the *Booster* module name, from which identifiers have to be imported in the annotation module. In this particular example, shape \( A \) and formal parameter \( X \) of the function *Matrix-Mult* are imported with the corresponding cardinality declaration. The mappings that are speci-
5.2 ANNOTATION LANGUAGE

fied here relate the shape and formal parameter to their representation in the
memory. For both data structures different linearisations are specified. In gen-
eral, mapping statements have the following syntax:

\[
\text{Mapping-statement} ::= \quad \text{Left-part} \leftarrow \text{Right-part} \\
\text{Left-part} ::= \quad \text{ViewShapeId}'[\text{free-var-exp}]' | \\
\quad \text{ViewFunction}(\text{ViewShapeId}) \\
\text{Right-part} ::= \quad \text{ProcMemName}'[\text{unconstrained-exp}, \\
\quad \text{free-var-exp}]' | \\
\quad \text{ViewFunction}($\text{ProcMemName}$)
\]

In contrast to view statements, functions with free variables may also be used
on the left hand side of the '<-' symbol. Note that for the dimensions of the
ProcMemName identifiers representing the memory and/or processor axis, an
unconstrained free variable expression is allowed.

Alternatives

The annotation language has a number of constructs that express a choice
between, or alternatives for annotations. The choice construct decides on basis
of, for example, the size of a shape which annotation to be chosen. The alternative
construct allows for the specification of multiple annotations and it is left to
the compiler to decide which one to choose.

The choice construct is:

\[
\text{IF} \; \text{Condition} \; \text{THEN} \; \text{mapping-statements} \\
\text{ELSE} \; \text{mapping-statements} \; \text{END}
\]

The alternative construct is:

\[
\text{ALTERNATIVE} \; \text{mapping-statements} \; \{ \; \text{OR} \; \text{mapping-statements} \; \} \; \text{END}
\]

5.2.5 Mapping and Interfacing to a Real Machine

Once the virtual machine has been defined and the relation with the identi-
fiers in the Booster program module has been specified, the programmer has
the option to relate the virtual machine to a real machine.

\[
\text{REAL MACHINE} \; \text{real-machine-name} (\text{processor-def} (\text{memory-def}) ); \\
\text{mapping-statements}
\]

The virtual machine, defined in terms of processors and memories, does not
necessarily match the underlying hardware. This is especially true when the
required number of virtual processors is only known at run-time. The mapping
from virtual to real machine can be influenced by the programmer, by specify-
ing the kind of scheduling or mapping technique that is to be applied. Consider,
for example, the virtual machine \texttt{vm} and the real machine \texttt{MyHardware}. We assume the following dimensions, where \( m \) is larger than \( p \):

\[
\text{VIRTUAL MACHINE } \texttt{vm} \ (\text{PROC } \texttt{vm\_proc} \ (m) ; \text{MEM } \texttt{vm\_mem} \ (x) \ (n)) ; \\
\text{REAL MACHINE } \texttt{MyHardware} \ (\text{PROC } \texttt{processor} \ (p) ; \text{MEM } \texttt{memory} \ (k)) ;
\]

\[
\texttt{vm\_proc}[i] \leftarrow \texttt{processor}[\text{FirstFree}(i)] ; \\
\texttt{vm\_mem}[i, j] \leftarrow \texttt{memory}[i*n + j] ;
\]

Here, \texttt{FirstFree} is some predefined scheduling routine that assigns at runtime a certain processor of the virtual machine \texttt{Proc} to the first processor of \texttt{MyHardware} ready to process more data. The \( x \) separate memories \texttt{vm\_mem} of the virtual machine \texttt{vm} are mapped onto the global memory of the real machine \texttt{MyHardware}.

### 5.3 Data Organisation

In our approach, an algorithm is defined in terms of shapes, which are in turn defined in terms of the memories of the underlying processor model. In this section, we particularly focus on the aspects of data organisation and disregard any implications it might have on the flow of control of the translated program on the target parallel machine.

The discussion on the data organisation concept will proceed as follows. First, we consider the \textit{representation} of data on a single — shared — memory. Next, we consider the \textit{partitioning} of data across a number of — non-shared — memories. Finally, we reflect on the organisation of the data on shared- and non-shared memories in time, that is in the evolution of the algorithm. This will be referred to as \textit{data incarnation} mappings.

#### 5.3.1 Data Representation Mappings

We define \textit{data representation} to be a mapping of the logical structure of the data — the shape — onto the memory of the virtual machine. This is especially relevant for multi-dimensional shapes, which can be represented on — linear — memory in many ways. It was shown amongst others in [Mace87], that for a number of architectures a different representation of the data structure in memory can have serious implications on the performance of the overall program.

A second reason for the need of data representation information, is the fact that various programs operate on shapes that only have a limited number of computationally relevant elements, as is the case for sparse matrix algorithms. It is very advantageous to exploit this additional information by storing only relevant elements and thus saving storage space. The annotation language knows two constructs to express this information. If the sparsity has a static nature, like band or triangular matrices, the representation information can be
expressed in terms of views in *Booster*. A shape is first viewed, within the *Booster* program itself or in the annotation module, and the resulting view is represented on some memory.

Below, mappings are discussed for each of the above described categories: mappings of shapes, mappings of views, and data-dependent mappings.

*Data representation mappings of shapes*

An example of such a representation is the following combination of a memory and a shape:

```plaintext
VIRTUAL MACHINE vm ( PROC processor (p); MEM M (n*m); );
IMPORT SomeAlgorithm :: A (n # m);
A[i,j] <- M[i*m+j];
A[i,j] <- M[i+j*n];
```

The first view statement defines a relation between the memory `M` and the shape `A`, much in the same way as relations between views and shapes are defined. In this case, `A` is considered as a dimension increasing view on the memory `M`. Effectively, it defines a row-wise mapping of `A` on `M`. The second view statement defines a column-wise mapping of `A`.

*Data representation mappings of views*

The representation of views allows for the description of partial changes in a shape's representation in the evolution of the program. The views that are represented can either be imported from the algorithm or new views can be specified in the annotation language to describe a desired mapping. For example, consider the following mapping statement in an annotation module:

```plaintext
VIRTUAL MACHINE vm ( PROC processor (p); MEM M (n); );
IMPORT SomeAlgorithm :: B (n);
B[2*i] <- M[i];
```

Of the sparse shape `B`, only the even elements are stored in memory `M`, not the odd ones. If in the algorithm the odd elements are accessed this will result in a run-time error.

An example of the use of view-representations is the forward and backward substitution step in the LU-algorithm discussed in Section 4.6.2. In the forward substitution step, a row-wise representation of `L` is preferable to any other representation and in the backward substitution phase, a column-wise representation of submatrix `U` is best. Both `L` and `U` are referring to shape `A` and therefore a single row- or column-wise representation of `A` does not suffice. The representation is best expressed in *Booster* in terms of the view identifiers `L` and `U`:
IMPORT LU-Factorisation :: L, U (n # n);
L[i,j] <- M[i*m+j];
U[i,j] <- M[i+j*n];

The semantics of these mappings are that through the views \( L \) and \( U \) the elements to which they refer in \( A \) are represented in memory such that in both cases they can be retrieved from consecutive memory locations. The above data organisation comes into effect just before the view identifiers \( L \) and \( U \) are used in the algorithm. Special attention has to be paid when representing expanding views. In that case the above specification may lead to a duplication of data. The views shown so far are all *structural* views, i.e. views that only depend on the structure of the shape, independent of its contents. *Data dependent* views are also very useful, as is demonstrated in the next subsection.

**Data dependent mappings:**

Certain types of algorithms are acting on sparse data structures. In the annotation language, we can use data dependent views to define how the sparse data structure is represented in memory, e.g.:

```
VIRTUAL MACHINE vm ( PROC processor (p); MEM M (n*m); );
IMPORT SomeAlgorithm :: A (n # m);
A[<> 0] <- M;
```

The content selection view is applied to \( A \), specifying all non-zero elements to be stored in memory. Note that this relation neither specifies the order in which the elements of the sparse matrix are to be stored, nor how many. When in the evolution of the algorithm the number of elements that are equal to zero changes, the number of elements stored in memory also changes.\(^3\) The efficient translation and integration of this (sparse) annotation is in general a difficult problem and will not be addressed any further in this thesis.

### 5.3.2 Data Partitioning Mappings

A generalisation of data representation mappings is the *data partitioning* mapping, that is the mapping of shapes and views onto different — non-shared — memories instead of a single shared memory. The notion of different, separate memories also introduces more degrees of freedom, e.g. the possibility to store a certain data item on more than one memory:

\(^3\) This can be the case when an algorithm generates fill-ins.
5.3 DATA ORGANISATION

VIRTUAL MACHINE vm (PROC proc(1); MEM memory (NrOfMem) (n));

IMPORT SomeAlgorithm::A (m); B (k);
A[i] <- memory[i mod NrOfMem, i div NrOfMem];
B[i] <- memory[0, i];

Prior to the import of shapes A and B, a virtual machine is defined, consisting of a memory structure memory with NrOfMem separate memories. When a virtual machine is defined with more than one memory the specification of the mapping of shapes or views needs to be extended with a precise specification of the required memory-modules. In the mapping statement, a memory consisting of several separate memories is denoted as a structure with an index set 0:NrOfMem-1 × 0:n-1, where the first dimension denotes the number of the memory module and the second the position of the specific data element in that memory.

The first mapping of the example relates shape A to memory M, where each element of A is assigned to a memory in a cyclic way. In contrast, the second mapping assigns shape B to the first memory in the memory structure only.

Sparse Structures
Data partitioning can also be defined on sparse structures. An example is the partitioning of a sparse data structure A, where we have the same virtual machine as in the previous example:

IMPORT SomeAlgorithm:: A (m);
SparseStorage[i] <- M[i mod NrOfMem, i div NrOfMem];
A[<> 0] <- SparseStorage;

Here the use of intermediate view identifiers in the annotation language is exemplified. The view identifier SparseA is first partitioned in a cyclic way on memory M. Subsequently, all non-zero elements of A are mapped on this view identifier SparseStorage, obtaining a consecutive storage for all non-zero elements of A. In turn, these consecutive elements are partitioned over a number of memories in a cyclic way.

Data Duplication
Data partitioning introduces the possibility to assign a certain data-item to multiple distinct memories. These kind of mappings are especially relevant to those architectures where data transfer rates between memories and processors form a bottle-neck in the computation. In the next section, more examples of the merits of these types of mappings will be given. Below we give a straightforward example of data duplication:
IMPORT SomeAlgorithm:: A (m);
A[i] <- M[_,i];
A[i] <- M[((i-1) mod NrOfMem):((i+1) mod NrOfMem),i];

The first mapping specifies a duplication of the vector A on every available memory in the system. In the second representation, the element A[i] is duplicated only on three consecutively numbered memories.

5.3.3 Data Incarnation Mappings

The semantics of imperative languages, like Fortran 77, C, and Pascal, define their variables to reside at a fixed location in memory. The values that are assigned to a certain variable are stored at that particular place in memory. An assignment to a particular variable will result in a re-use of the memory location occupied by the previous value. We will refer to this as re-use memory-assignment behaviour.

The semantics of single-assignment languages, like SISAL [Sked85] and Id Nouveau [Arvind88], forbid the re-use of memory. In principle, for each new value of a variable — also referred to as incarnation — a new place in memory is allocated. Although smart compilers for single-assignment languages can introduce some form of memory re-use behaviour without violating semantics, it cannot be influenced by the programmer. The latter point also holds for imperative languages: memory-assignment behaviour is fixed. The programmer can only influence this by altering the algorithm itself.

This brings us to the essence of effective portability of programs for parallel machines. Memory-assignment behaviour can strongly influence the amount of parallelism available in a program. A programmer — or compiler for that matter — should have the possibility to influence this behaviour without any need for changes in the algorithm itself.

Therefore, Booster's data organisation concept offers the possibility to define data incarnation mappings. In this concept, each data-item, be it a shape or a view, has an incarnation-number associated with it. This incarnation-number indicates how many times a certain data-item has been modified. This number can in turn be used in data organisation mappings to define any memory behaviour, ranging from re-use to single-assignment. Examples of this mapping are given below:

VIRTUAL MACHINE vm (PROC processor (1); MEM M (k) (n));

IMPORT SomeAlgorithm:: B (n);
B[i] <- M[INC*n + i];
B[i] <- M[(INC mod 2)*n + i];
B[i] <- M[i];
The incarnation-number is indicated by the standard keyword \texttt{INC}. In this example, the first mapping defines a single assignment memory behaviour: for each incarnation \( k \), a unique location is defined. The second mapping defines an alternating scheme between two memories. The third and last mapping is the default: \textit{re-use memory assignment behaviour}. The three mapping schemes are illustrated in Figure 19. The top most illustrates the single assignment (1), the middle one (2) the wrap re-use, and the lower-one (3) the re-use memory assignment.

![Figure 19 Memory assignment behaviour]

\textit{Interference of mappings}

The use of multiple views of the same shape in the annotation module, may lead to contradictions in the specified mappings. For example, if two identity views are taken on the same shape, but mapped differently on the virtual machine, this either leads to a duplication of the shape or to an emulation of one of the mappings. Neither of these options is desirable and therefore the \textit{collateral} annotation of views and shapes is not permitted, unless it does not appear in the same fragment of the annotation, or the specified views are not overlapping. The constraint cannot, in all cases, be checked at compile-time and is in those cases deferred to run-time.

An example is the LU-factorization annotation in Section 5.3.1, where view identifiers \( L \) and \( U \) refer to the same shape \( A \), but they refer to distinct regions of \( A \). It is therefore permitted to use both annotations.

\subsection{5.4 Computation Organisation}

The data organisation concept as introduced in the previous sections is used to define where data should reside at a given moment in the evolution of the algorithm. This, however, leaves the issue of the processor responsible for the computation of data indeterminate. Data organisation alone is not enough to specify any given execution behaviour.\footnote{For example, if we assume as default that a processor nearest to a certain memory is responsible for the processing of all data in that memory, we would still have a problem with shared-memory machines. Furthermore, when duplicating data by partitioning the data in some way, this does not necessarily imply a duplication of processing responsibility.} Therefore, we introduce the notion of
**computation organisation:** the assignment of responsibility for certain computations to (possibly distinct) processors.

Computation and data organisation have a comparable categorization of mappings. Data organisation was subdivided in *data representation, data partitioning*, and *data incarnation* mappings. Their equivalents in computation organisation are: *computation representation, computation partitioning*, and *computation incarnation* mappings, respectively. *Computation representation* defines the sequentialization of potential parallel computations. *Computation partitioning* distributes the responsibility for the computation over a number of processors. Finally, *computation incarnation* defines a responsibility mapping that may change in time. In this thesis, we will restrict ourselves to the description of computation partitioning and we will not elaborate on computation representation and incarnation mappings.

### 5.4.1 Computation Responsibility Convention

Computation partitioning distributes responsibility of computations over a number of processors by assigning computation responsibility to processors. An example of this is a virtual machine `vm` that contains a processor structure `processor` with `p` processors, on which shapes `A` and `B` are decomposed:

```plaintext
VIRTUAL MACHINE vm (PROC proc(p); MEM memory (n div p));

IMPORT SomeAlgorithm: A, B (n);
A[i] <- proc[i div (n div p)];
B[i] <- proc[i mod (n div p)];
```

This computation partitioning mapping results in the assignment of the responsibility for element `i` of `A` to processor `i div (n div p)`. Note that this partitioning results in a *block-wise* assignment to the processor structure. Likewise, responsibility for shape `B` is assigned to processor `i mod (n div p)`. This partitioning results in a *cyclic* assignment of responsibility to the processor structure.

**Convention**

If we relate an annotation to a *Booster* program, we observe that a certain computation (expression) may involve more than one shape, resulting in several possible responsibility assignments. For example consider the *Booster* program fragment:

```plaintext
SHAPE A, B (n) OF INT;
A := B + 2;
```
Without a convention it is not clear where, that is by which processor, the addition is to be performed. To overcome this ambiguity, we introduce the following convention:

The processor responsible for the element to which a certain expression is assigned is responsible for the computation of the whole expression.

According to this convention, the processor responsible for \( A[i] \) is also responsible for the computation of \( B[i] + 2 \).

### 5.4.2 Computation Partitioning Mappings

A first example of the computation partitioning mapping has already been given in Section 5.4.1. These mappings are very similar to data partitioning mappings except for their semantics: partitioning of computation responsibility rather than data. The syntax of both types of mappings is equal and we will therefore focus our attention to some example applications of this mapping.

*Computation Partitioning Mappings on Views*

Computation partitioning mappings, like data organisation mappings, may also be defined on views. A view, unlike a shape, does not have a fixed size and may change during computation. Consequently, any computation partitioning defined on such a view also changes. This is illustrated in the following *Booster*- and corresponding annotation module.

```plaintext
MODULE Comp (A) -> (A);
A (n) OF INT;
BEGIN
  V <- A;
  WHILE SIZE(V) > 0 DO
    V := V*2;
    V <- V[1:upb];
  END;
END;

ANNOTATION MODULE Comp ;

VIRTUAL MACHINE vm (PROC proc(p));

IMPORT Comp:: V (m)
  V[i] <- proc[i div (m div p)];
END.
```

On the left side, a *Booster* module `Comp` describes an algorithm, where the view identifier \( V \) initially refers to shape \( A \). In the while-statement, the view identifier is redefined in terms of itself, resulting in a shrinking view. The computation partitioning description on the right side, defines a block-wise partitioning of \( V \) on the processors of virtual machine \( vm \). As the partitioning is defined in terms of the size of the view identifier \( V \), it will change after each redefinition of \( V \) in the view statement.
5.4.3 Integrating Computation and Data Organisation

The definition of computation- and data organisation has for reasons of clarity been treated separately in the previous sections. These cases, however, are just special cases of the general virtual machine mappings that describes computation as well as data organisation. Consider,

\[
\text{VIRTUAL MACHINE } \text{vm} (\text{PROC } \text{processor} \ (p), \ \text{MEM } \text{memory} \ (m) \ (k)); \\
\text{IMPORT } \text{SomeAlgorithm}:: \ A \ (n); \\
A[i] \leftarrow \text{vm}[f(i), \ g(i), \ h(i)];
\]

Here, the general template contains the relations \(f(i)\) and \(g(i)\) which denote the computation responsibility and memory assignment, respectively. The function \(h(i)\) defines the ordering of the elements within a single memory. This annotation description on a virtual machine is equivalent to the two following expanded processor- and memory-structure mappings:

\[
A[i] \leftarrow \text{processor}[f(i)]; \\
A[i] \leftarrow \text{memory}[g(i), \ h(i)];
\]

For distributed-memory architectures, data may be assigned to memories that are not nearest to the processor responsible for processing of this data. This obviously results in additional communication needed to obtain data.

On some machines processors and memories come in pairs (when preceded by the keyword \text{PROCMEM}) and here the assignment of responsibility and memory can be combined by using a single relation \(f\), e.g.

\[
\text{VIRTUAL MACHINE } \text{vm} (\text{PROCMEM } \text{proc} \ (p) \ (k)); \\
\text{IMPORT } \text{SomeAlgorithm}:: \ A \ (n); \\
A[i] \leftarrow \text{vm}[f(i), h(i)];
\]

The function \(h\) defines the ordering of the data within the memory.

5.5 Exploiting Parallelism

The concept of programming for parallelism with annotations is best explained by a number of example annotations of Booster programs. In these examples, we will devote our attention to techniques to obtain minimal execution-time. These techniques include load-balancing, that is: the even spread of computations over all processors in time, and optimisation of data transport. For optimisation of data transport we distinguish between techniques to obtain optimal transport between memories on the same level as well as between memories in the memory-hierarchy.
5.5 EXPLOITING PARALLELISM

5.5.1 Load-balancing

The main objective of parallel programming is to minimise total execution time. If we disregard data transport and synchronisation-time, the best way to achieve this is to assign responsibility such, that the amount of (computational) work is distributed evenly over the available processors in time. This technique is referred to as load-balancing. How load-balancing is achieved depends on the properties of algorithm and the architecture on which it is to be executed. We can, however, distinguish two general categories: compile-time or run-time load-balancing.

Compile-time load-balancing

At compile-time, we have two forms of load-balancing: static- and dynamic load-balancing. Static load-balancing is the assignment of computation responsibility to the virtual machine such that each processor has a constant work-portion that is fixed at the start of the algorithm.

Dynamic load-balancing is the adaptation of responsibility for computation in the evolution of the algorithm. If we relate this to Booster, regions of the basic data structures involved in processing may change. With this change of regions, the workload for individual processors may change too, causing unbalanced execution behaviour. For some algorithms this change of regions is predictable, which enables the programmer to attain load-balancing at compile-time by defining a dynamic partitioning to redistribute execution responsibility among processors.\(^5\)

Examples

Static and dynamic load-balancing are best illustrated by an example. To this purpose, consider the Successive Over Relaxation (SOR) and the Gaussian elimination algorithm of Section 4.6. (The latter as a specialisation of the LU-factorization algorithm). In these examples, we do not make any distinction between shared- or distributed architectures. We merely illustrate the principle of load-balancing, while disregarding the effects of data transport and synchronisation overhead.

Static Load-balancing

For convenience, we recapitulate the significant part of the SOR-algorithm:

\(^5\)This possibly at the expense of additional communication
SHAPE A (n # n) OF REAL;
ITER k DO
    A[interior] := SOR(A,c);
END;

In the following annotation module, this program is now related to a virtual machine VM. The mappings of the shape A of SOR are defined with the help of the view functions row_partitioning and diagonal_part.

ANNOTATION MODULE SOR;

VIRTUAL MACHINE VM (PROC processor (p), MEM memory (n));

VIEW FUNCTION row_partitioning (VM) -> (X);
VM( q # k ), X (n);
BEGIN
    X[i]<-VM[i div (n div p), i mod (n div p)];
END;

VIEW FUNCTION diagonal_part (VM) -> (X);
VM ( q # k ), Y (n # m);
BEGIN
    Y[i,j] <- VM[(i+j) div ((n+m-1) div p),
                   (i-j) + m*((i+j) mod ((2*n-1) div p))];
END;

IMPORT SOR::A;
ALTERNATIVE
    A <- row_partitioning(VM); OR
    A <- diagonal_part(VM);
END;

Above, two alternatives for the mapping of shape A are defined: either a row_partitioning or a diagonal mapping. Diagonal partitioning is a refinement that can exploit vector-capabilities of an architecture.

Dynamic load-balancing

The Gaussian elimination algorithm, unlike the SOR-algorithm, does not use all elements of data structure A in each iteration. The algorithm is defined in terms of views which decrease in size each iteration. We therefore have to devise another partitioning to obtain load-balancing.

Let us recall the Gaussian elimination algorithm and adapt it somewhat by adding a view identifier H, referring to the union of the regions P, R, C, and B. This additional view identifier allows us to define the required partitioning.
Note that this additional view identifier does not alter the semantics of the program.

\[
\begin{align*}
H & \leftarrow A; \\
\text{WHILE} & \quad \text{SIZE}(H) > 1 \quad \text{DO} \\
(P,R,C,B) & \leftarrow \text{Pivoting}(H); \\
B[i,j] & \leftarrow B[i,j] - (C[i]/P)*R[j]; \\
H & \leftarrow H[1:\text{upb},1:\text{upb}]; \\
\text{END};
\end{align*}
\]

The computational work is performed solely on regions of \( A \), referred to by the view identifier \( H \). The view identifier \( H \) decreases in size — one row and column — in each repetition step of the \textsc{While}-statement. A row-wise partitioning on the shape \( A \) itself would therefore eventually result in processors to idle. A row-wise partitioning in terms of this view identifier \( H \), however, results in a more balanced version:

\[
\begin{align*}
\text{IMPORT} & \quad \text{GaussianElim::}H \quad (k \neq k); \\
H & \leftarrow \text{row\_partitioning}(\text{VM});
\end{align*}
\]

The static partitioning versus the dynamic are depicted in \textit{Figure 20.a} and \textit{20.b}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig20.png}
\caption{\textit{Static row-wise partitioning} \quad \textit{20.b Dynamic row-wise partitioning}}
\end{figure}

\textbf{Run-time load-balancing}

For some algorithms, the load at compile-time cannot be predicted because the amount of time needed for the computation is dependent on the values of the input data. As a consequence, we have to decide at run-time which computation is to be executed on which processor, dependent on the current load of the computing system. Needless to say that this run-time element introduces a certain amount of overhead. This overhead can not be avoided because it is inherent to the algorithm itself: the appropriate information is only available at runtime.

In the annotation language, the programmer can influence the load for each processor by influencing the mapping of the virtual machine onto the real machine. Usually, a virtual machine is created that consists of a large number
of processors and the algorithm is decomposed on this machine. In turn, this virtual machine is mapped on the real machine according to a scheduling strategy defined by the programmer. The various strategies were introduced in Section 5.2.5.

5.5.2 Data Management

The techniques for load-balancing discussed in the previous section, suffice for shared-memory parallel machines. On distributed-memory machines and shared-memory machines with memory hierarchies, however, a nicely decomposed algorithm can have a very poor performance due to communication delays. Careful data management is therefore important when considering parallel programming on these machines.

To overcome this problem, the programmer can add data partitioning and data incarnation mappings to influence the data allocation and the data transport between memories. For example, a programmer can influence the communication behaviour of the system by data or computation duplication, which can — depending on the algorithm, the replicated data, and the computation—lead to more computational work, but less communication. The opportunity to make a trade-off between computation and communication is especially of importance for systems that have a small network bandwidth, yielding long network delays.

Transport to and from the different stages in the memory hierarchy can also be described. For example, data incarnation mapping can be used to define where data should reside at any given moment during program execution. These concepts are best illustrated with examples:

*Red/Black algorithm*

We illustrate the use of data duplications to limit the amount of communication with the *Red/Black* algorithm of Section 4.6. Recall that this algorithm subdivides the input matrix like a chess-board, but with a one-element overlap. Consider the mapping where the algorithm is executed on a distributed-memory machine, where each block is assigned to a different processor and the data-blocks are stored locally.

```
ANNOTATION MODULE RedBlack;

VIRTUAL MACHINE vm ( PROC processor (p), MEM m (n));

IMPORT RedBlack:: R, B (n [#] n);
R <- row_partitioning(vm);
B <- row_partitioning(vm);
END.
```
The elements of \( R \) and \( B \) are mapped to the virtual machine block-wise for each row. This mapping defines a duplication of computation responsibility of all borders of the blocks. For this particular program all data elements are now available locally for computation.

**Simplex method**

Another illustration of data management in the case of distributed-memory architectures is the optimisation data transport for the Simplex algorithm. We observe that the regions \( T[\upb,1:\upb] \) and \( T[1:\upb,0] \) — corresponding to the optimisation vector \( C \) and the constraint vector \( B \) — are needed by every processor to compute the new pivot elements for each iteration. A plain row-wise partitioning and assignment of the template \( T \) to a virtual machine, would result in the redistribution of these regions during every iteration, yielding overhead in communication. The duplication of these regions limits the amount of communication at the expense, however, of additional storage. This is formalised as follows:

```plaintext
ANNOTATION MODULE Simplex;

VIRTUAL MACHINE vm (PROC processor (p), MEM memory (n));

IMPORT Simplex::T (n ≠ m);
T[0:upb-1,1:upb] ← row_partitioning(vm);
T[upb,1:upb] ← processor;
T[1:upb,0] ← processor;
END.
```

### 5.5.3 Fine-tuning of Algorithms

In order to obtain efficient executing programs on specialised hardware, we need the interplay between algorithm design and annotations. As an example, we will describe various versions of a matrix multiplication algorithm amongst others mapped to a shared-, distributed-memory and a systolic (parallel) machine. To this purpose, we first consider a general description of the algorithm, which then gradually evolves to a more specific and efficient algorithm.

The first step in the description process is the following straightforward description of multiplication in *Booster*:

\[
C[i,j] := \text{reduce}(+,A[i,\_]*B[\_,j]);
\]

A possible annotation module is the following:
CHAPTER 5 EXPLOITING THE PARALLEL VIRTUAL MACHINE

ANNOTATION MODULE MatrixMult;

VIRTUAL MACHINE vm ( PROC proc (n); MEM mem (n) (n \# n);)

IMPORT MatrixMult::A, B, C;

C[i, _] <- proc[i];
C <- mem[0, _, _];
A <- vm[0:n-1, _, _];
B <- vm[0:n-1, _, _];
END.

The responsibility for the computation associated with row i of matrix C is assigned to the i-th processor, and, initially, shapes A and B are stored in the memories. The resulting data (in shape C) is stored in the memory associated with machine-number 0.

Although the above algorithm and annotations permit the description of the initial and final data and computation organisation, the algorithm itself has no facilities to describe the intermediate organisation of data and computation. The fine-tuning of the algorithm is especially of importance when the underlying architecture has some form of memory hierarchy or a specific control structure as is found in systolic systems.

Fine-tuning of data transport

Further tuning of the algorithm entails the introduction of an intermediate view identifier T, that refers to a 3-dimensional structure resulting from the multiplication of the matrices.

\[
T[i, j, k] \leftarrow A[i, k] \times B[k, j];
C[i, j] := \text{REDUCE}(+, T[i, j, _]);
\]

Note that the introduction of the view identifier does not introduce additional overhead in terms of storage. The variable T just provides a hook to be able to describe more precise data and computation organisations as is demonstrated in the following annotation module:

ANNOTATION MODULE MatrixMult2;

VIRTUAL MACHINE vm ( PROC processor (n \# n);
MEM memory (n) (k));

IMPORT MatrixMult::T (m \# m \# m);
ALTERNATE
T[i, j, _] <- processor[i, j, _];
OR
T[j, _, i] <- processor[i, j, _];
END
END.
5.5 EXPLOITING PARALLELISM

The two alternate mappings are visualised in Figure 21. The box depicts view identifier T and the numbers on the subboxes denote the processor responsible for processing of the elements in that particular subbox.

![Figure 21 The first and second version of the mapping algorithm](image)

**Fine-tuning for Systolic real machines**

For the next version of the algorithm, we consider the mapping of a further refined matrix multiplication algorithm for a systolic architecture. Systolic architectures differ from other more general parallel machines, in their more restricted control-structure and interconnection topology. In the annotation module, we want to be able to describe where in each stage of the algorithm data resides and computations take place. Therefore, we refine the original algorithm by introducing view identifiers S, T, and R on shapes A, B, and C, respectively.

```
VIEW FUNCTION rotate (A) -> (B);
A, B (n+1);
BEGIN
    B[i] <- A[(i+1) mod n];
END;
```

(1) \( R <- C; \)
(2) \( T[i,j] <- B[j,i]; \)
(3) \( \text{ITER n DO} \)
(4) \( R[i,j] := R[i,j] + A[i,j] \ast T[0,j]; \)
(5) \( T <- T[\text{rotate,}_]; \)
(6) \( R <- R[\_, \text{rotate}]; \)
END;

The corresponding annotation module is:
The variable \( i \) varies from 0 to \( n-1 \) and increases every time the rotate view is taken on \( T \). From the placement of \( T \), we can derive the placement of \( B \) through the relation: \( T_{ij} \) "refers to" \( B_{ji} \). The flow of data through the systolic array is derived by moving data from the previous storage-place to the next.
Chapter 6
The View Calculus

In this chapter, we will discuss a formal basis for both Booster and the annotation language. Although this formal basis is used for Booster, it can also be applied to other languages than Booster. The actual translation of Booster to this formal basis will be described in Chapter 7. The integration with annotations and the generation of parallel programs will be the topic of Chapter 8, although the necessary preparations are already being made in this chapter.

6.1 Description of semantics

Many theories and approaches have been suggested for the specification of the semantics and properties of programs in a certain language. The main approaches are operational- [Pagan81], denotational [Tennent76, Tennent81], and axiomatic or algebraic [Bergstra89] semantics. The reader is referred to [Meyer90] for a practical survey and comparison. Operational and denotational semantics are closer to the implementation of interpreters and compilers, respectively, whereas axiomatic semantics approaches are often used to prove certain properties of a program, such as correctness. For the specification of semantics of parallel or concurrent systems theories like, Petri nets [Reisig85], high-level operational semantics [Hoare78], [Hoare85], and algebraic techniques [Milner89] have been suggested.

Goals

The choice of a formalism to describe the semantics of a particular language depends on what needs to be specified and what is to be derived from that specification. In our case we aim at four goals:

• The formalism must be independent of any annotation or machine architecture;
• The formalism must accommodate the description of performance enhancing transformations;
• Specification of the temporal behavior resulting from an annotated Booster program must be possible;
• All semantic specifications must have a close correspondence to the implementation of a compiler that realises all steps.
Options
The above goals strongly suggest a denotational approach to the semantics of Booster, because of the close correspondence to the compiler implementation and the independence of a specific machine architecture. The approach should, however, allow for the description and integration of the annotations within the same framework.

In addition, the formalism describing the semantics of Booster and its annotations should also allow for the specification of performance enhancing transformations that are dependent on the target architecture. To this purpose a calculus, that also guarantees the correctness of transformations, would be most convenient.

Finally, the semantics of annotated Booster-programs have to be defined in terms of a formalism capable of expressing the temporal aspects of co-operating processes. This formalism should be close to the programming interface provided by current parallel machines, i.e. the union of shared-memory and message-passing programming models.

Choices
We have defined a single formalism to satisfy the above goals. The formalism is a calculus referred to as V-cal, which is short for View-calculus\(^1\). V-cal is used to express the semantics of Booster and serves as an intermediate formalism on which a number of transformations are defined in the form of rewrite-rules. The semantics of V-cal itself are described in terms of function- and set-theory. V-cal also contains machine-dependent features, which serve as a basis to describe the temporal aspects of Booster and its annotations. From this extended formalism the notions of performance and efficiency used in the performance enhancing transformations are derived.

Summary
In summary, to achieve the aforementioned goals we have chosen for the V-cal formalism to describe the semantics of Booster and its annotation language. V-cal consists of three parts:

- Expressions in V-cal are described in a denotational form based on set- and function-theory;
- The state-to-state semantics of V-cal are described in denotational form;
- The temporal and data transport behaviour of V-cal is described in terms of a synchronisation and communication model;

\(^1\) This name was first introduced by Arjan van Gemund, whose master thesis [Gemund89] has been the starting point for the work reported in this chapter.
Other Semantic Formalisms

We have not chosen to formulate the semantics of Booster in terms of high-level specification languages such as VDM [Björner82] or Z [Spivey88], because these formalisms are not very well suited for the description of rewrite-rules that are extensively used in the translation and optimisation process. ASF-SDF [Bergstra89] allows for the description of the semantics of Booster, but for optimisations a formalism like V-cal is more appropriate. Although developed independently, V-cal resembles the Bird-Meertens [Bird87] formalism. The Bird-Meertens formalism focuses on algorithmic description involving lists, whereas V-cal focusses on operations on array-type of data structures.

Overview of the chapter

In the section Basic Concepts, we introduce the basic notions from set- and function-theory and derive the Basic Formalism needed for the semantic description of V-cal. The relationship between Booster, V-cal, Basic Formalism, Transformations, and Optimisations and the sections in which they are described are depicted in the following figure:

![Figure 22. Organisation of Chapter 6 and relation with Chapter 7.](image)

Section 6.3 describes V-cal in terms of the concepts introduced in Section 6.2. In Sections 6.3 and 6.4, a number of transformations on V-cal are defined. In Section 7.1, Booster programs are translated to V-cal. Finally, in Section 7.2, transformations in terms of reduction-rules on V-cal-expressions are defined.

6.2. Basic Concepts and Notation

For the basic mathematical concepts and notations used in the remainder of this thesis, only the following principal definitions have been included.

6.2.1. Index Sets

The index set within V-cal is a central notion for which we have chosen a specific representation to be elaborated on in the sequel of this chapter. To define index sets, we define a restricted version of the Cartesian product: the
bounded Cartesian product. For completeness, we include the definition of the
genral Cartesian product.

Definition 6.1

The Cartesian product of $d$ arbitrary sets $A_1, \ldots, A_d$ ($d \geq 1$) is a set of vectors
defined as follows:

$$A_1 \times \ldots \times A_d = \{ (a_1, \ldots, a_d) \mid a_i \in A_i, i = 1, \ldots, d \}$$

We denote the vector $(a_1, \ldots, a_d)$ as $a$. If $A_i = A$ for all $i$, then the shorthand
notation for the Cartesian product is: $A^d$.

To our purposes the Cartesian product is used to define an arbitrary, but fixed
number of finite sets. This is formalised as follows, where $N$ denotes the posi-
tive natural numbers including zero.

Definition 6.2

Let $l = (l_1, l_2, \ldots, l_d)$ and $u = (u_1, u_2, \ldots, u_d)$, with $l, u \in N^d$, then a bounded
Cartesian product is defined as the Cartesian product of sets $N_i \subset N$,
where

$$N_i = \{ n \mid l_i \leq n \leq u_i \}$$

$i = 1, \ldots, d$

We denote this as $N_1 \times \ldots \times N_d$ or $N^b$, where $b = (l, u)$. The positive integer
$d$ is the dimension of $N^b$, vector $l$ is its lowerbound, $u$ is its upperbound,
and the last two together form a pair of bounds, $b$. This $b$ will be referred to
as the bounding vector. The Cartesian product $N^b$ is referred to as a
bounded set. Note that $N^b \subset N^d$.

We will follow the notational convention that all bounded Cartesian products
have a superscript in bold, whereas the Cartesian product of $d$ sets $N: N^d$ is de-
noted in normal format. Having defined the bounded Cartesian product, we de-
fine a projection function on such a set, yielding an element in a specified
dimension.

Definition 6.3

Let $N^b$ be a bounded set of dimension $d$, then the projection function $\pi: N \times
N^b \to N \cup \{ \bot \}$ is defined as:

$$\pi(k, (a_1, \ldots, a_d)) = \begin{cases} a_k & \text{if } 1 \leq k \leq d \\ \bot & \text{if } k < 1 \lor k > d \end{cases}$$
where \( \bot \) denotes the *undefined* symbol.

We use the following shorthand notation: \( \pi_k(a) \), or just \( \alpha_k \) if no ambiguity arises. Furthermore, we need a notion of equivalence of elements (vectors) within the \( \mathcal{N}^b \) space.

**Definition 6.4**

Let \( x, y \in \mathcal{N}^b \), where \( d \) is the dimension of \( b \), then:

\[
\begin{align*}
x = y & \Rightarrow \forall i \in 1, \ldots, d \; x_i = y_i \\
x \neq y & \Rightarrow \neg(x = y)
\end{align*}
\]

**Ordinal Function**

For several purposes, we need an *ordering* on (subsets of) bounded sets. We have chosen for the following total — lexicographic — ordering on the finite subset \( S \subseteq \mathcal{N}^b : (S, \prec) \).

**Definition 6.5**

Let \( S \subseteq \mathcal{N}^b \) and \( \prec \) be a relation on \( S \), then the ordering \((S, \prec)\) is defined, with \( i, j \in S \) and \( i \neq j \), as:

\[
i \prec j \iff \exists k \in \{1, \ldots, d\} \{ i_k < j_k \land \forall 1 \leq n \leq k-1 \; i_n = j_n \}
\]

Because \( S \subseteq \mathcal{N}^b \) is a finite set of tuples of natural numbers and it holds for every \( i, j \in S \) with \( i \neq j \) that \( i < j \) or \( j < i \), the ordering is indeed total. We can thus define a *monotonic total* function \( \text{ordinal} : \mathcal{N}^d \to \mathcal{N} \), which assigns a unique number to each element of \( S \).

**Definition 6.6**

Let \((S, \prec)\) be a total ordering, then the function \( \text{ordinal} : \text{pow}(\mathcal{N}^d) \times \mathcal{N}^d \to \mathcal{N} \) on \( S \) is defined as follows:

\[
\begin{align*}
(1) \; \forall i, j \in S : (i < j \land \exists k \in S : i < k < j) & \iff \text{ordinal}_S(j) = \text{ordinal}_S(i) + 1 \\
(2) \; (\exists k \in S : k < i) & \Rightarrow \text{ordinal}_S(i) = 0
\end{align*}
\]

The first restriction defines \( \text{ordinal} \) to preserve the ordering from index sets to the set of natural numbers. The second restriction formalises the fact that the ordinal function numbers from zero onwards.
The position function
Based on the aforementioned ordinal function, we introduce its inverse function position. This function takes two arguments as input: a subset of a bounded set and an ordinal position and returns the associated element of the set at the desired position.

Definition 6.7
Let \( \text{pos} : \text{pow}(N^d) \times N \rightarrow N^d \), where \( S \) is a finite set:

\[
\text{pos}_S(p) = i, \text{ where } i \in S, \text{ordinal}_S(i) = p, \text{ and } p = 0, \ldots, |S| - 1.
\]

Because the function ordinal\(_S\) is total, the above set contains exactly one element for each \( p \) in \( 0, \ldots, |S| - 1 \). Hence, \( \text{pos}_S \) is a total monotonic increasing function in \( p \). In addition, we define a number of primitive operations on vectors.

Definition 6.8
For arbitrary vectors \( a, b \subseteq N^d \), we can define the following "element-wise" addition, subtraction, minimum, and maximum of vectors \((N^d \times N^d \rightarrow N^d)\)

\[
\begin{align*}
a \pm b &= (a_1 \pm b_1, \ldots, a_d \pm b_d) \\
\min(a, b) &= (\min(a_1, b_1), \ldots, \min(a_d, b_d)) \\
\max(a, b) &= (\max(a_1, b_1), \ldots, \max(a_d, b_d))
\end{align*}
\]

Furthermore, we define the zero-vector, which exists for each value of \( d \):

\( \theta = (0, \ldots, 0) \)

When the intersection of sets defined by bounded Cartesian products is to be computed, this can be derived directly from the bounding vectors. To this purpose, the operator ' & ' on bounding vectors is defined. The operator is apart from vectors with elements from \( N \) also defined for two additional elements \( (\bot, \infty) \). Here \( \bot \) is the undefined symbol, or error, and \((0, \infty)\) the unity element of the operation. We introduce the following denotation for the domain of bounding vectors:

\[
B^d = (N^d \times N^d) \cup \{ \bot, (0, \infty) \}
\]

Definition 6.9
For arbitrary bounding vectors \( x, y \in N^d \times N^d \), which are used in bounded Cartesian products, the following intersection operator ' & ' : \( B^d \times B^d \rightarrow B^d \) is defined: \( x = (x_l, x_u) \) and \( y = (y_l, y_u) \):
\[ x \& y = \begin{cases} 
(\max(x_i,y_i):\min(x_u,y_u)) & \text{if } \forall \ i \ y_i \leq x_i \leq y_u, \forall x_i \leq y_i \leq x_u \\
\bot & \text{otherwise}
\end{cases} \]

\((0,\infty) \& x = x, \ x \& (0,\infty) = x, \ (0,\infty) \& (0,\infty) = (0,\infty)\)

\(\bot \& x = \bot, \ x \& \bot = \bot, \ \bot \& \bot = \bot\)

We will often denote bounding vectors in a more explicit form than \(a = (l,u)\) by writing: \(l_1:u_1 \times l_2:u_2 \times \ldots \times l_d:u_d\). In the following definition the set \(T\) is the set of truth values: \{True, False\}.

**Definition 6.10**

An index set \(I\) is defined by a bounded set \(N^b\), on which a predicate function \(P : N^b \rightarrow T\) is defined:

\[ I = \{ i \in N^b \mid P(i) \} \quad \text{or short-hand} \quad I = (b,P). \]

**Example 6.1**

Consider, the index set \(I = (a,P)\), with bounding vector \(a = (l,u)\), where \(l = (1,2)\) and \(u = (3,4)\), or alternatively \(a = 1:3 \times 2:4\). Let the predicate be defined as follows \(P((i_1,i_2)) = (i_1 < i_2)\). This effectively restricts the elements in the index set to the upper right triangle of a matrix. The same index set can be defined by the following index set \(J = (b,Q)\), with bounding vector \(b = 0:9 \times 0:9\) and predicate \(Q((i_1,i_2)) = (i_1 < i_2 \land 1 \leq i_1 \leq 3 \land 2 \leq i_2 \leq 4)\). Both \(I\) and \(J\) describe the same index set, but \(J\) has a larger bounded set than \(I\). We are specifically interested in minimal bounded sets and we therefore define the notion of a normalised index set.

**Definition 6.11**

A normalised index set \(I = ((l,u), P)\) is an index set of dimension \(d\) for which the following constraint holds:

\[ \min_{i \in I}(i_k) = l_k \land \max_{i \in I}(i_k) = u_k \quad \quad k = 1, \ldots, d \]

In the sequel, we will refer to a normalised index set as index set if no ambiguity arises. The cardinality of an index set is defined as follows:

**Definition 6.12**

The cardinality vector \(c\) of a normalised index set \(I = ((l,u), P)\) is defined as:

\[ c = u - l + 1 \]
For some index sets the lowerbound is equal to zero and we will refer to these sets as zero-based normalised index sets.

**Definition 6.13**
A zero-based normalised index set \( I = ((l,u), P) \) is a normalised index set for which:
\[
l = 0
\]

Note that the zero-based normalised index set can be characterised by the cardinality vector only, because \( c = u - l + 1 = u + 1 \). In addition, the following useful functions are defined on index sets.

**Definition 6.14**
- \( \text{Size: } \text{Pow}(\mathcal{N}^d) \to \mathbb{N} \) returns the size of an index set: \( \text{size}(I) = |I| \);
- \( \text{Dim: } \text{Pow}(\mathcal{N}^d) \to \mathbb{N} \) returns the dimension of an index set: \( \text{dim}(I) = d \);
- \( \text{Card: } \text{Pow}(\mathcal{N}^d) \to \mathcal{N}^d \) returns the cardinality of an index set: \( \text{card}(I) = c \)

### 6.2.2. Data Structures and Views

After the preliminary introduction of index sets, we can now introduce the notion of a data structure:

**Definition 6.15**
A data structure \( S \) is a tuple \( (A, D, I, f) \), where
1. \( A \) is the data set of the data structure with \( A \in \{ \mathbb{Z}, \mathbb{R}, \mathbb{T} \} \);
2. \( D \) is the set of data values: \( D \subseteq A \);
3. \( I \) is a zero-based normalised index set;
4. \( f : \mathcal{N}^d \to A \) is a data function, that is a total function such that \( f(I) = D \)

The functions \( \text{size} \), \( \text{dim} \), and \( \text{card} \) are also relevant for data structures and are equal to the corresponding functions for the index set of the data structure. For example, the size of a data structure is equal to the size of the index set of that structure.

**Views**

Data structures can be accessed via views by means of a view application. A view is an onto and one-to-one relation that is defined on index sets. In general, relations are much harder to define and manipulate than functions and therefore a representation has to be found such that the desired manipulations on these relations can be essentially described in terms of functions. Given the fact that views are relations that are defined on index sets represented by a set
comprehension, we can use this additional information in representing a view. The representation of index sets is made up by a tuple of a bounded rectangular set and a predicate function on that set. The same tuple is found in the representation of a view, which is represented as \((K, dp, ip)\). This tuple consists of a bounding index set \(K = (b, R)\) and two total functions \(ip\) and \(dp\), called the index propagation function, and the domain propagation function, respectively. Whenever convenient we will use the quadruple notation \((b, R, dp, ip)\). Formally, the view is defined as follows:

**Definition 6.16**

A view \(V \subseteq I \times J\) is an onto and one-to-one relation described by the quadruple \((b, R, dp, ip)\), with index sets \(I = (x, P)\) and \(J = (y, Q)\), where \(d = \text{Dim}(x)\), \(e = \text{Dim}(y)\). The function:

\[
 dp : B^d \times (B^d \rightarrow T) \rightarrow B^e
\]

defines the bounding vector \(y = b \& dp(x, P)\) for the domain of the function \(ip\) and predicate \(R\). In addition,

\[
 dp(\perp, P) = \perp \text{ and } dp(((0, \infty), (0, \infty)), P) = ((0, \infty), (0, \infty)).
\]

Furthermore, the \(dp\) function must have the following property, where \(U\) and \(W\) are bounded index sets:

\[
 dp(U \cap W) = dp(U) \& dp(W)
\]

The function \(ip\), with signature \(N^x \times N^y \rightarrow N^x\), defines the actual relation between the elements of \(J\) and \(I\). Finally, the predicate \(R : N^y \rightarrow T\) restricts the number of elements that are accepted by a view. A view defines the following relation between \(I\) and \(J\):

\[
 J = (y, Q) = (b \& dp(x, P), (P \circ ip) \land R)
\]

Furthermore, the function \(ip\) restricted to \(J\): \(ip(I)\mid_J\) is a total function from \(J\) to \(I\).

The relation between the index set \(I = (x, P)\) and \(J = (y, Q) = (b \& dp(x, P), (P \circ ip) \land R)\) is to be understood as follows. The resulting bounding vector \(y\) is formed by taking the intersection of the bounding vectors \(b\) and \(dp(x, P)\), where the bounding vector \(b\) expresses the static 'filter' property of a view. The dynamic 'filter' property of the view depends on the bounding vector of the input index set and is input for the domain propagating function \(dp\). The same holds for the predicate \(Q\), which is made up by the view's 'static' predicate \(R\) and the
inputs predicate $P$ after application of the index propagation function $ip: (P \circ ip)$.  

The above illustrates the name domain propagation for $dp$, because in part this function determines the size and shape of the codomain of the view $V$ depending on the domain of the view. Along the same line, the name index propagation function refers to the fact that $ip$ defines a relation between the indices of the codomain and domain of a view $V$.

It is easy to see that the quadruple representation does indeed define an onto and one-to-one relation between $I$ and $J$. The onto property is a direct consequence of the fact that $ip$ is a total function from $J$ to $I$, which implies that for each element of $J$ there is at least one element in $I$. Furthermore, from the fact that $ip_J$ is a function, we can derive that for each element $j$ in $J$ there is at most one element $i$ in $I$ such that $(i,j) \in V$. Hence, the relation $V$ is also one-to-one.

The domain propagation function takes both a bounding vector as well as a predicate as an argument. In practice, for most views the bounding vector itself is sufficient. For reasons of simplicity, we will from now on restrict ourselves to the domain propagation function which only has a bounding vector as argument.

**Simple versus Complex Views**

We distinguish between two types of views: simple and complex views. These views differ with respect to the "domain" of index and domain propagation functions. In the above definition the domain propagation function is defined on $dp : B^d \times (B^d \rightarrow T) \rightarrow B^e$ with the property $dp(U \cap W) = dp(U) \& dp(W)$ and $ip: N^x \times N^y \rightarrow N^x$. We will refer to this as a complex view.

A simple view has a domain propagation function $dp : B^d \rightarrow B^e$, with the property $dp(a \& b) = dp(a) \& dp(b)$, where $a$ and $b$ are bounding vectors. Moreover, the index propagation function has a signature $ip: N^x \times N^y \rightarrow N^x$. Note that the simple view is a special case of the complex view.

**Example 6.2**

An example of a simple view $V = (b,R,dp,ip)$ where:

- $b = (0,\infty)$, $R(i) = True$, $ip(i) = 2*i$, $dp((l,u)) = (l \ div \ 2, u \ div \ 2)$

The bounding index set $K = (b,R)$ is trivial in this case and the view is completely defined by the index- and domain-propagating functions. This view selects all even elements ($ip$) and consequently propagates ($dp$) a domain that is half the size of the original.

Another example of a simple view is the following selection view:

- $b = (1,4)$, $R(i) = \{ 2*i \geq 1 \}$, $dp((l,u)) = (l,u)$, $ip(i) = i$
The view is now fully determined by the bounding index set \( K = (b,R) \) and the
codomain \( J \) is independent of the input \( I \). These examples illustrate the duality
of these specific representations for views.

An example of a complex view is the following view that takes an index set as
input and renumbers it to obtain a consecutive index set:

\[
b = (0,\infty), \quad R(i) = \text{True}, \quad dp(I) = 0:size(I) - 1, \quad ip_f(i) = pos_f(i)
\]

Both \( ip \) and \( dp \) need the input index set \( I \) to define the view. The domain propagation function returns the size of the input index set and the index propagation function maps the consecutive range of elements in the index set on their originals using the previously defined \( pos \) function.

**View Application**

A view \( V \) can be related to a data structure \( S \) by a view application denoted as
\( V(S) \). This application means that the index set \( I = (a,P) \) of the data structure \( S = (A,D,I,f) \) is the input index for the view \( V = (b,R,dp,ip) \). As a consequence, the
access to the resulting data structure is defined as \( f(ip_f(i)) \in J \) for \( i \in V(I) \),
where \( J = (b \& dp(a,P), P \circ ip \wedge R) \).

The view in the view application \( V(S) \) can be reduced to a normal form. We
will need this notion in the sequel of this chapter.

**Definition 6.17**

A view \( V = (b,R,dp,ip) \) in a view application \( V(S) \), where \( S \) is a data struc-
ture with index \( I = (a,P) \), is in normal form iff:

\[
J = (b,R)
\]

According to this definition, the index set that results from a view application
in normal form is equal to the bounding index set of the view, and for these type
of views the domain propagation function can be eliminated.

**View Composition**

The essence of simple views is that they form a group that is closed under
composition. This composition can be performed independent of the argument
the composition is eventually applied to. For complex views this is not true: they
may only appear as the rightmost element of a composition.

We will need the following lemma that states that a composition of domain
propagation functions is again a domain propagation function.
Lemma 6.1

Let \( dp_s : B^e \to B^f \) and \( dp_c : B^d \times (B^d \to T) \to B^e \) be domain propagation functions of simple and complex views, respectively, and \( U \) and \( V \) be index sets, then:

\[
\forall U, V \in B^d \times (B^d \to T) : dp_s(dp_c(U \cap V)) = dp_s(dp_c(U)) \land dp_s(dp_c(V))
\]

Proof:

As \( dp_s \) and \( dp_c \) are domain propagation functions they have the properties \( dp_c(U \cap V) = dp_c(U) \land dp_c(V) \) and \( dp_s(\alpha \& b) = dp_s(\alpha) \land dp_s(b) \), respectively. We can now derive the following:

\[
dp_s(dp_c(U \cap V)) = dp_s(dp_c(U) \land dp_c(V)) = dp_s(dp_c(U)) \land dp_s(dp_c(V))
\]

\[\Box\]

A special case of lemma 6.1 is the composition of two simple domain propagation functions \( dp \) and \( dp' \). This again yields a simple domain propagation function, because:

\[
dp'(dp_s(\alpha \& b)) = dp'(dp_s(\alpha) \land dp_s(b)) = dp'(dp_s(\alpha)) \land dp'(dp_s(b))
\]

The Theorem below states that the composition of a simple and a complex view is a complex view.

Theorem 6.18

Let \( V = (b_v, R_v, dp_v, ip_v) \subseteq H \times J \) be a simple view and \( W = (b_w, R_w, dp_w, ip_w) \subseteq I \times H \) be a complex view then the result of the view composition \( V \circ W \subseteq I \times J \) is a complex view \( U = (b, R, dp, ip) \), where:

\[
\begin{align*}
b &= b_v \& dp_v(b_w), \\
R &= R_v \& (R_w \circ ip_v), \\
\text{dp} &= dp_v \circ dp_w, \\
\text{ip} &= ip_w(\circ \circ ip_v).
\end{align*}
\]

Proof:

Consider an arbitrary, but fixed index set \( I = (x, P) \). By applying view \( W \) to \( I \) we obtain, according to Definition 6.16 the index set \( H \):

\[
H = (b_w \& dp_w(x, P), (P \circ ip_w(I)) \& R_w)
\]  \hspace{1cm} (1)

Application of \( V \) to index set \( H \) yields, again according to Definition 6.16, the following index set:

\[
J = (b_v \& dp_v(b_w \& dp_w(x, P)), R_v \& ((P \circ ip_w(I)) \& R_w) \circ ip_v))
\]  \hspace{1cm} (2)
The set of bounding vectors of index set (2) can be rewritten, using the property of the \( dp \) function as:

\[
\mathbf{b}_v \& dp_v(b_w \& dp_w(x,P)) = b_v \& dp_v(b_w) \& dp_v(dp_w(x,P))
\]  

(3)

Note that by renaming \( b = b_v \& dp_v(b_w) \) and \( dp = dp_v \circ dp_w \) we obtain the form: \( b \& dp(x) \), for (3). The new function \( dp = dp_v \circ dp_w \) according to Lemma 6.1 again has the domain propagation function property.

Similarly, the predicate expression for index set (2) can be rewritten as:

\[
R_v \wedge ((P \circ ip_w(I)) \wedge R_w) \circ ip_v = R_v \wedge (R_w \circ ip_v) \wedge (P \circ ip_w(I) \circ ip_v)
\]  

(4)

By renaming \( R = R_v \wedge (R_w \circ ip_v), ip = ip_w(I) \circ ip_v \), we obtain the form \((P \circ ip(I)) \wedge R\) for (4).

As a result the composition of two simple views is also proved, because the complex view is a generalisation of the simple view. Moreover, the composition of two simple views also yields a simple view, because the composition of two simple domain propagation functions is simple.

Through the definition of view composition, the rationale for representing a view as a quadruple becomes clear. The composition of views can now be expressed as an intersection of their bounding index sets, a composition and combination of predicates, and compositions of the index- and domain-propagation functions, yielding again a view with the same representation. There is however a restriction; complex views may not be composed with simple views: only the other way around. This restriction is inherent to the dependence of the complex view on its input index set. In the next chapter, we will elaborate on the practical implications of this restriction.

**Example 6.3**

We illustrate the principle of view composition with an example:

Let \( V = (b_v, R_v, dp_v, ip_v) \)

\( b_v = (0:2), R_v(i) = \{ i \geq 1 \} \), \( dp_v(l,u) = (l,u - 2) \), \( ip_v(i) = i + 2 \)

\( W = (b_w, R_w, dp_w, ip_w) \)

\( b_w = (0:10), R_w(i) = \{ i \geq 2 \} \), \( dp_w(l,u) = (l,u \text{ div } 2) \), \( ip_w(i) = 2*i \)

Then \( V \circ W = (b_v \circ_w, R_v \circ_w, dp_v \circ_w, ip_v \circ_w) \):

\( b_v \circ_w = (0:2) \& (-2:10-2) = (0:2) \),
\( R_v \circ_w(i) = \{ i \geq 2 \} \circ ip_v \wedge \{ i \geq 1 \} = \{ i + 2 \geq 2 \} \wedge \{ i \geq 1 \} = \{ i \geq 1 \} \),
\( dp_v \circ_w(l,u) = (l, (u \text{ div } 2) - 2) \),
\( ip_v \circ_w(i) = 2*(i + 2) = 2*i + 4 \).
In Section 6.3, it will become clear that views are often applied recursively. Therefore, we define the $k$-composition of views as follows:

**Definition 6.19**

$V^k$ is the $k$-composition of $k$ identical views, where

$$V^k = \begin{cases} id & \text{if } k = 0 \\ V \circ V^{k-1} & \text{if } k > 0 \end{cases}$$

where $id$ denotes the identity function: $id(i) = i$.

\[ \square \]

Note that the $k$-composition is only defined for those views that have input- and output-index sets of the same dimension. Given the definition of the $k$-composition $V^k$, we can derive the following result, if we assume that the predicate of the view equals $True$.

**Theorem 6.20**

Let $V = (b, True, dp, ip)$ be a simple view, where $dp(b) \subseteq b$, then its $k^{th}$ composition equals:

$$V^k = (dp^{k-1}(b), True, dp^k, ip^k)$$

**Proof:**

Proof by induction on $x$. For $k = 1$: $V^1 = (b, True, dp, ip) = (dp^0(b), True, dp, ip)$

Assume that for $k-1$ the assumption holds, then:

$$V^k = V \circ V^{k-1}$$

According to **Definition 6.18** this composition is equal to:

$$V^k = (b, True, dp, ip) \circ (dp^{k-2}(b), True, dp^{k-1}, ip^{k-1}) = (b \& dp^{k-1}(b), True, dp^k, ip^k)$$

If we consider the bounding vector of $V^k$, then $b \& dp^{k-1}(b) = dp^{k-1}(b)$. This is because $dp(b) \subseteq b$, then $dp^2(b) \subseteq dp(b)$, if for $k-1$ the assumption holds $dp^{k-1}(b) \subseteq b$, then $dp(dp^{k-1}(b)) \subseteq dp^{k-1}(b) \subseteq b$.

By induction, it is clear that the bounding vector of $V^k$ equals $dp^{k-1}(b)$.

$$V^k = (dp^{k-1}(b), True, dp^k, ip^k)$$

\[ \square \]

### 6.2.3. Parameterized Index Sets and Relations

The index sets and relations introduced in the previous sections do not depend on any value that is defined outside their scope. In $V$-cal, however, the
concept of parameterization is necessary. This is illustrated by the following
definition of a view \( V = (b, R, dp, ip) \):

\[
\begin{align*}
    b &= (x,y), \quad R(i) = \{ i \leq k \}, \quad dp(i) = i \div n, \quad ip(i) = n^*i.
\end{align*}
\]

Here, \( b \) has parameters \( x \) and \( y \), \( R \) has a parameter \( k \), and \( dp \) and \( ip \) have a parameter \( n \). In this section, we define the notion of parameterized index sets and relations more formally. In addition, we will consider a number of specific functions, such as projections on parameterized relations and index sets.

**Parameters**

A parameter is a finite string over some alphabet, and the set of parameters \( \text{Par} \) is defined as: \( \text{Par} \subseteq \Sigma^* \), where \( \Sigma = \{ a, b, c, \ldots, z \} \) is a finite alphabet. An example of a subset of a parameter set is: \( \{ a, aabb, abc, j \} \subseteq \text{Par} \).

**Parameterized Relation**

The parameterized relation is a relation from a set of parameters and a set to a set:

\[
V \subseteq (\text{pow}(\text{Par}) \times N^d) \times N^e
\]

We will denote a parameterized relation as: \( V_x \). An example of such a relation is the following function: \( f_x(i) = i \div x \). This function is curried with respect to the parameter \( x \). It is clear that the previous definitions and theorems are equally valid if parameterized relations are involved.

**Parameterized Index Set**

The application of parameterized views to (non-parameterized) index sets yields an index set that potentially depends on parameters. We therefore introduce the concept of parameterized index sets. The chosen representation of the index set dictates that if \( I = (x,P) \) is parameterized, this implies that either \( x, P \) or both are parameterized. Consider the following parameterized index sets:

\[
I = ((n,n-1), \{ i \leq m \}) \quad \text{and} \quad J = ((2, n \div 2), \{ k \leq i \leq m \})
\]

A parameterized index set is denoted by subscripting it with the parameter or parameter-set, yielding for example \( I_{n,m} \) and \( J_{k,n,m} \).

This concludes the definitions and theorems of the Basic Formalism. This apparatus will enable us to form \( V\text{-cal} \), which is the subject of the next section.

### 6.3. \( V\text{-CAL} \)

The view calculus, \( V\text{-cal} \), serves three purposes. First, it is used to describe the semantics of the \( \text{Booster} \)-language. Secondly, \( V\text{-cal} \) is used as an intermediate formal system for the description of performance enhancing transformations on \( \text{Booster} \) programs. Finally, \( V\text{-cal} \) will enable us to deduce criteria for
those index sets and view functions that produce efficiently running programs, when translated to the virtual machine. This categorization of index sets and view functions lies at the heart of the optimisation of (parallel) translations. \( V\text{-cal} \) itself is described in terms of the definitions given in the previous section.

This section is organised as follows. The syntax of \( V\text{-cal} \) is defined in Section 6.3.1, where we make a distinction between (view-)expressions and clauses. This is followed in Section 6.3.2. by the introduction of reduction rules for view-expressions. Reduction rules on clauses are elaborated on in Section 6.3.3.

6.3.1. Structure of \( V\text{-cal} \)

The view calculus can be subdivided in three parts. The first part of \( V\text{-cal} \) is the view expression domain. The semantic basis for this part has been given in the previous section, where the notions Index sets, Views, Data Structures, and Projections were introduced. In the \( V\text{-cal} \) context, we introduce another set of elements: Bindings, Selections, Reductions, and Special Functions which are syntactic extensions based on the previous notions. These extensions do not change the semantics, but are chosen such that one can express various optimisations and translations. The second part of \( V\text{-cal} \) are the Clauses, which include Assignment, Repeat and Conditional statements and Orderings between collections of these statements. The third and last part of \( V\text{-cal} \) consists of machine related constructs such as synchronisation and data transport operations. These three parts of \( V\text{-cal} \) are depicted in Figure 23.

![Figure 23 The structure of V-cal](image-url)
View expressions and clauses will be described in this chapter. The machine related part of V-cal will be elaborated on in Chapter 8. The semantics of the language related part are again subdivided in (view-)expressions and clauses.

6.3.2 View Expressions

Figure 23 already revealed some of the structure of view expressions in V-cal. View expressions are defined in terms of the three basic notions introduced in Section 6.2:

\[ \sqrt{I, dp, ip} \quad A \text{ view with an index set } I = (\mathcal{B}, \mathcal{R}); \]

A Datastructure or View identifier;

\[ \pi_{\text{range}} \quad A \text{ projection that selects a number of dimensions of its argument, according to the dimension numbers specified in the set range;} \]

f A scalar function;

The scalar function f includes functions like addition, subtraction, multiplication, etc. The abstract syntax of view expressions is as follows:

\[
ve = A \\
\quad | \sqrt{I, dp, ip} \, ve \\
\quad | f(ve_1, ve_2, \ldots, ve_n) \\
\quad | (\pi_{x_1}(ve), \ldots, \pi_{x_n}(ve)) \quad \text{where } \forall i, j : x_i \cap x_j = \emptyset
\]

The extensions

These four basic notions are not sufficient to meet the requirements formulated in the introduction to this chapter. We therefore need to extend the view expression part of V-cal. For convenience, we recall the requirements involved:

- We want to specify the semantics of Booster independent of any annotation or machine architecture;
- We want to be able to specify performance enhancing transformations and prove the correctness of these transformations;
- All semantic specifications must have a close correspondence to the implementation of a compiler that realises all steps.

The first requirement has been met in part by introducing, amongst others, the view concept. The second nor the third requirement, however, is met by the view concept, because it only gives an functional description of the semantics. We therefore have to find a representation that is semantically equivalent to the view concept, but allows for the description and implementation of perfor-
mance enhancing transformations. In addition, it must be possible to define metrics on view expressions that define some notion of performance.

**Extensions to V-cal**

To meet the above requirements, we introduce an instantiation of and an extension to the V-cal representation. Prominent in V-cal is the notion of a view, which is used to define a relation between index sets. To capture the specifics of certain hardware elements, however, we need to reformulate the view in a selection and generation part. This instantiation can be used to define notions like vector-operations, reductions, and selection functions. The additional elements of V-cal are introduced below. Their semantics are discussed in the next section.

**Instantiation**
The selection and generation, or parameter binding, part of a view are denoted as follows:

\[
[f(i)] \quad \text{A selection, where } f(i) \text{ is a function in (a vector of parameters) } i;
\]

\[
\Delta(i \leftarrow I) \quad \text{A parameter binding 'binds' a vector of parameters } i \text{ to an index set } I;
\]

**Extension**
Based on the four basic forms of view expressions and the parameter binding, we define higher order functions that represent certain coherence between operations.

\[
\#^d \quad \text{Superscalar functions of dimension } d;
\]

\[
\text{reduce}(\#, \Delta(i \leftarrow I)) \quad \text{The reduce operation with a binary function } \# \text{ and a parameter binding.}
\]

Examples of superscalar functions are element-wise vector or matrix operations. These superscalar functions also include selection functions, such as the max function, and reduce functions such as the sum of the values of a vector.

**Syntax**
The abstract syntax for view expressions is extended as follows, where <old-ve> denotes the already existing elements of V-cal:

\[
ve = \quad <\text{old-ve}>
\]

\[
\| \Delta(i \leftarrow I) ve
\]

\[
\| \text{reduce}(f, \Delta(i \leftarrow I)) ve(i)
\]

\[
\| [f(i)] ve
\]

\[
\| \#^d(ve_1(i), ..., ve_n(i))
\]
When the parameter of a parameter binding does not occur in the view expression it is bound to, this parameter is referred to as free, i.e. \(\Delta(i \leftarrow I) [J]A\), then \(i\) is free. If the parameter of the parameter binding does occur in the view expression this is referred to as bound, i.e. \(\Delta(i \leftarrow I) [I]A\), then \(i\) is bound. Note that every bound parameter binding in \(i\) has a selection \([f(i)]\), a parameterized parameter binding \(\Delta(j \leftarrow I(i))\) or \(\text{reduce}(f, \Delta(j \leftarrow I(i)))\) associated with it.

**Semantics of Additional V-cal Elements**

The view-concept introduced in Section 6.2 forms the basis for the parameter binding and selection definitions:

\[
\begin{align*}
\Delta(i \leftarrow I) \Delta(j \leftarrow J) [f(i,j)] &= \sqrt{(I \times J, id, f)} \\
[f(c)] &= \sqrt{(c \times c), (id, f)}
\end{align*}
\]

As a consequence, the application of the parameter binding and selection pair to an index set \(X\) yields the same index set \(Y\) as the application of the view on \(X\). The difference with the view construct is that parameter bindings and selections are distinct entities. This is, for example, convenient in combination with *superscalar functions*, which essentially define element-wise operations on multi-dimensional structures. The denotational definition for these functions is as follows, where \(\#^e\) with \(e = 0\) is the scalar function and \(e = \text{dim}(K)\):

\[
\sqrt{(K, dp, ip)} \#^d(ve_1, ve_2, ..., ve_n) = \#^e(\sqrt{(K, dp, ip) ve_1, \sqrt{(K, dp, ip) ve_2, ..., \sqrt{(K, dp, ip) ve_n}}})
\]

Hence, the application of a view to a superscalar function equals a superscalar function of a different dimension. When \(|K| = 1\) and \(ip = x\), we may rewrite this form as follows according to (2) and (3):

\[
[x] \#^d(ve_1, ve_2, ..., ve_n) = \#^e([x] ve_1, [x] ve_2, ..., [x] ve_n)
\]

If we apply (4) to a (*superscalar*) vector multiplication \(*^1\), we obtain the following definition for a vector operation in terms of a scalar operation:

\[
[x] *^1(A, B) = *([x]A, [x]B)
\]

**Reduce**

For other types of computations, such as the sum of a vector of numbers, we introduce the reduction or *reduce* construct. The *reduce* is defined recursively too, where \(I \cap J = \emptyset, I \cup J = K\), and \(|K| \geq 2\):

\[
\text{reduce}(\#, \Delta(k \leftarrow K)) ve(k) = (\text{reduce}(\#, \Delta(k \leftarrow I)) ve(k)) \# (\text{reduce}(\#, \Delta(k \leftarrow J)) ve(k))
\]

The sum of a vector and matrix is defined as follows:
\[ \text{reduce}(+, \Delta((i,j) \leftarrow 0:1 \times 0:1)) [i,j] B = ([0,0]B + [0,1]B + [1,0]B + [1,1]B) \]

There are two exceptions to the above reduce-rule. If the index set only contains
one element, this element is substituted in the expression and returned. If the
index set is empty this results in an empty expression:

\[ \text{reduce}(\#, \Delta(k \leftarrow \{e:e\})) ve(k) = ve(e) \]
\[ \text{reduce}(\#, \Delta(k \leftarrow \emptyset) ve(k) = \emptyset \]

Choice functions

Choice functions are superscalar functions for the retrieval of elements
using some criterium such as the maximum or minimum. In V-cal, the
binary functions \text{min}, \text{max}, \text{minindex}, and \text{maxindex} are available. The latter
two are defined on the domain of \((\text{IndexSet} \times \text{Value})^2 \rightarrow \mathbb{N}\), whereas the first
two are defined on the domain \(\text{Value}^2 \rightarrow \text{Value}\). Examples are:

\[ \text{max}(A,B) \text{ and } \text{minindex}(C,D) \]

A special case of the \text{choice} functions is the unary, multi-dimensional version.
In this case the \text{choice} function is applied to a single multi-dimensional argument and
yields a single value. An example is the 1-ary \text{max} function applied
to a data structure \(A\) with index \(I\):

\[ \text{max}(A) = \text{reduce}(\text{max}', \Delta(k \leftarrow I)) [k]A \]

where \(\text{max}'\) is a binary scalar function. As a consequence unary choice functions
can be expressed in terms of a binary scalar function and a reduce.

Summary

The above definitions are summarised in the table below, where each of the
definitions is referred to by an identifier \('V'\) followed by a number. This \('V'\)
indicates the view expression domain on which these definitions are given. In addition,
each definition is referred to by a short name, such as \text{view instantiation}.

<table>
<thead>
<tr>
<th>Nr</th>
<th>Definition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td><strong>View Instantiation</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\Delta(i \leftarrow I) \Delta(j \leftarrow J) [\ell(i,j)] \Leftrightarrow \sqrt{(I \times J, id, f)})</td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td><strong>View Simplification</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>([\ell(c)] \Leftrightarrow \sqrt{(c:c), id, f)})</td>
<td></td>
</tr>
</tbody>
</table>
### 6.3 V-CAL

<table>
<thead>
<tr>
<th>V3</th>
<th>Superscalar functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sqrt{(K, , dp, , ip)} #^d(ve_1, , ve_2, , ..., , ve_n) \iff #^e(\sqrt{(K, , dp, , ip)} , ve_1, , \sqrt{(K, , dp, , ip)} , ve_2, , ..., , \sqrt{(K, , dp, , ip)} , ve_n) )</td>
<td>( e = \text{Dim}(K) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V4</th>
<th>Reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{reduce}(#, , \Delta(k \leftarrow I \cup J)) , ve(k) \iff )</td>
<td>( I \cap J = \emptyset, )</td>
</tr>
<tr>
<td>( (\text{reduce}(#, , \Delta(k \leftarrow I)) , ve(k)) , # )</td>
<td>(</td>
</tr>
<tr>
<td>( (\text{reduce}(#, , \Delta(k \leftarrow J)) , ve(k)) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V5</th>
<th>Special reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{reduce}(#, , \Delta(i \leftarrow L)) , ve(i) \iff \left{ \begin{array}{l} \text{ve}(c) \quad L = {c:c} \ \emptyset \quad L = \emptyset \end{array} \right. )</td>
<td>( \emptyset # , \emptyset = \emptyset )</td>
</tr>
</tbody>
</table>

#### Index function for view expressions

The notion of index sets is central in V-cal and for some operations we need to derive the index set of a certain view expression. We not only need this notion to determine whether certain index sets match, but also for data dependence analysis on V-cal programs as will be introduced in Section 6.4. The index function is defined as follows:

**Definition 6.21**

Let \( \text{Index} : \text{Pow(ViewExpression)} \to \text{Pow(IndexSet)}, \text{Index}(ve) = (a,P), \) and \# a superscalar function, then:

\[
\begin{align*}
\text{Index}(\sqrt{(b, \, R, \, dp, \, ip)} \, ve) &= \text{Def} \, 6.16 \; (b \, \& \, dp(a), \, R \, \wedge \, P \, \circ \, ip); \\
\text{Index}(\text{DataStructure}) &= \text{DataStructure.Index}; \\
\text{Index}(\#(ve_1, \, ve_2, \, ..., \, ve_n)) &= 0; \\
\text{Index}(\pi_x(ve)) &= \pi_x(\text{Index}(ve))
\end{align*}
\]

Definitions V1 to V5 allow us to derive the index function for the extensions of V-cal too:

\[
\begin{align*}
\text{Index}(\Delta(k \leftarrow K) \, [f(k)] \, ve) &= V1 \; \text{Index}(\sqrt{(K, \, True, \, id, \, f)} \, ve) = K \\
\text{Index}(\{i\} \, ve) &= V2 \; \text{Index}(\sqrt{(i:i)} \, True, \, id, \, id) \, ve) = \text{Def} \, 6.16 \; (i:i); \\
\text{Index}(\#^e(ve_1, \, ve_2, \, ..., \, ve_n)) &= \\
&= \text{Index}(\#^e(\sqrt{(K, \, True, \, id, \, id) \, ve_1, ..., \, \sqrt{(K, \, True, \, id, \, id) \, ve_n}})) = \\
&= \text{Index}(\sqrt{(K, \, True, \, id, \, id) \, \#(ve_1, ..., \, ve_n)}) = K = \text{Index}(ve) \\
\text{Index}(\text{reduce}(\#^e, \, \Delta(k \leftarrow K)) \, ve(k)) &= \text{Index}(ve(i)), \text{ where } i \in K.
\end{align*}
\]
Example 6.5
Consider the following examples, where $\text{Index}(A) = \text{Index}(B) = K$ and $\ast^1$ a superscalar multiplication:

\[
\text{Index}(\Delta(k \leftarrow K) [k] A^1 [k] B) = \text{Index}(\Delta(k \leftarrow K) [k] A^1 B) = K
\]
\[
\text{Index}(A \ast^1 B) = \text{Index}(A) = K
\]
\[
\text{Index}(\max^1(A)) = 0
\]

6.3.3 View Expression Reductions
The definitions $VI$ to $V6$ describe the semantics of various (compositions of) language elements of $V$-cal. From these definitions, we can derive theorems that yield further relations between $V$-cal language elements. An example of such a derivation is the contraction of parameter bindings and selections. This contraction can be derived using the equivalence between a pair of parameter bindings and selections with views ($VI$), which allows us to translate Theorem 6.20 on view composition into an equivalent in terms of parameter expressions and selections:

\[
\sqrt{\langle i, id, f \rangle} \sqrt{\langle J, id, g \rangle} = \text{Def 6.18} \sqrt{\langle K, id, g \circ f \rangle}
\]

\[
\sqrt{\langle i, id, f \rangle} \sqrt{\langle J, id, g \rangle} = ^{VI} \Delta(i \leftarrow I) [f(i)] \Delta(j \leftarrow J) [g(j)]
\]

\[
\sqrt{\langle K, id, g \circ f \rangle} = ^{VI} \Delta(i \leftarrow K) [g(f(i))]
\]

Other interactions between elements in $V$-cal have been summarised in the Table II and corresponding proofs are given in Appendix A.

Table II Derived theorems for extended $V$-cal

<table>
<thead>
<tr>
<th>Nr</th>
<th>Derived Theorem</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V6$</td>
<td><strong>Parameter Decomposition</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\Delta(j,k) \leftarrow J \times K [f(j,k) \rangle \vee \leftrightarrow$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\Delta(j \leftarrow J) \Delta(k \leftarrow K) [f(j,k) \rangle \vee$</td>
<td></td>
</tr>
<tr>
<td>$V7$</td>
<td><strong>Selection Composition</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[i] [j] \Leftrightarrow [i]$</td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td>Simplification</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>Δ(ι → I) [ι] ve ⇔ ve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index(ve) = I</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V9</th>
<th>Extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[f(ι), g(ι)] ⇔ Δ(x → (b, (x = f(ι)))) [x, g(ι)]</td>
<td></td>
</tr>
<tr>
<td>f(ι) ∈ Nb</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V10a</th>
<th>Contraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[g(ι), ] Δ((j,k) → J × K) [f(ι), k] ⇔ Δ(k → K) [f(g(ι)), k]</td>
<td></td>
</tr>
<tr>
<td>g(ι) ∈ J</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V10b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[g(ι), ] Δ((j,k) → J × K(j)) [f(k)] ⇔ Δ(k → K(g(ι)))) [f(k)]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V11</th>
<th>Functions and selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ι] #d(ve_1, ve_2, ..., ve_n) ⇔ #e([ι]ve_1, [ι]ve_2, ..., [ι]ve_n)</td>
<td></td>
</tr>
<tr>
<td>e = d − Dim(ι)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V12</th>
<th>Functions and Parameter Bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ(k → K) #d(ve_1, ..., ve_n) ⇔ #e(Δ(k → K) ve_1, ..., Δ(k → K) ve_n)</td>
<td></td>
</tr>
<tr>
<td>e = d + Dim(K) for i ∈ {1, 2, ..., n}: ve_i = [h(k)] ve'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V13</th>
<th>Views and Reduces</th>
</tr>
</thead>
<tbody>
<tr>
<td>∨(I, dp, ip) reduce(#e, Δ(k → K)) ve(k) ⇔ reduce(#d, Δ(k → K)) ∨(I, dp, ip) ve(k)</td>
<td></td>
</tr>
<tr>
<td>e = Dim(ip) and d = Dim(I) + e</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V14</th>
<th>Selections and Reduces</th>
</tr>
</thead>
<tbody>
<tr>
<td>[j] reduce(#d, Δ(k → K)) ve(k) ⇔ reduce(#e, Δ(k → K)) [j] ve(k)</td>
<td></td>
</tr>
<tr>
<td>e = d − Dim(j)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V15</th>
<th>Parameter Bindings and Reduces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ(ι → I) reduce(#d, Δ(k → K)) ve(ι,k) ⇔ reduce(#e, Δ(k → K)) Δ(ι → I) ve(ι,k)</td>
<td></td>
</tr>
<tr>
<td>e = Dim(I) + d</td>
<td></td>
</tr>
</tbody>
</table>
### Example 6.9 Using the reductions

We will present a number of examples that illustrate the application of the above derived theorems on example expressions derived from programs.

**Interactions with Superscalar functions**

The incorporation of a parameter binding in a function or operator increases its dimension. For example, a scalar operator \( \cdot \) changes to a vector operator \( \cdot^1 \).

\[
\Delta(k \leftarrow 0:10) ([k]A \cdot [k]B) = V12 (\Delta(k \leftarrow 0:10) [k]A) \cdot^1 (\Delta(k \leftarrow 0:10) [k]B)
\]

**Interactions with reduces**

In Section 4.6.1 an example of the computation of the trace of a matrix, that resulted from a matrix multiplication, was given. This program translates to V-cal as follows and can be reduced (Here the constructs in bold are involved in the reduction step)

\[
\text{reduce}(*, \Delta(x)) [x,x] \text{reduce}(*, \Delta(k)) [i,j] ([i,k]A \cdot [k,j]B)) = V14 \\
\text{reduce}(*, \Delta(x)) \text{reduce}(+, \Delta(k)) [x,x] \Delta(i,j) ([i,k]A \cdot [k,j]B)) = V12 \\
\text{reduce}(*, \Delta(x)) \text{reduce}(+, \Delta(k)) [x,x] \Delta(i,j) ([i,k]A \cdot^2 [k,j]B)) = V11 \\
\text{reduce}(*, \Delta(x)) \text{reduce}(+, \Delta(k)) [x,x] \Delta(i,j) ([i,k]A \cdot [x,x] \Delta(i,j) [k,j]B)) = V10 \\
\text{reduce}(*, \Delta(x)) \text{reduce}(+, \Delta(k)) ([x,k]A \cdot [k,x]B))
\]

The effect of the reduction is the integration of the selection of the diagonal with the matrix multiplication. This example illustrates the result of a contraction on a view expression and yields a result close to intuition: a view expression that requires the least number of operations to be performed. We will elaborate on these optimisations in the next chapter.

Consider the following example where \( \cdot^1 \) is a superscalar vector multiplication operator:
\[ \text{reduce}(\ast^1, \Delta(i \leftarrow I)) (\Delta(j \leftarrow J) ((i,j)A + [j]B)) = V6 \]
\[ \Delta(j \leftarrow J) \text{reduce}(\ast, \Delta(i \leftarrow I)) ((i,j)A + [j]B)) \]

The effect of the application of theorem V6 is the reformulation of the reduction with a vector operator in terms of a vector of scalar reductions.

**Generalised Contraction**

The contraction defined in Theorem V10(a, b) needs three terms: a selection, a parameter binding, and again a selection. In most practical cases, we want to contract selections and parameter bindings that are not directly followed by a selection. In example 6.9, the reduction of the program fragment that described the trace of a matrix showed that the current Theorem V10, requires a number of intermediate steps (V12 and V11) to obtain a form for which V10 applies. Consequently, we want to change Theorem V10 such that:

**Theorem Generalised Contraction**

Let \( \Delta(j \leftarrow J) \) be a bound parameter binding, where \( g(i) \in J \), then

\[ [g(i)]) \Delta(j \leftarrow J) ve(j) \Rightarrow ve(g(i)) \]

**Proof:**
The proof of this generalised contraction is obtained by induction. We exchange the selection \([g(i)]\) and the parameter binding \( \Delta(j \leftarrow J) \) with the element to the right until we encounter a selection. We can then apply one of the Theorems V10 a or b. Note that every parameter binding has at least one selection or parameterized parameter binding associated with it to the right and that consequently this process is finite if the view expression is finite.

Hence, we have to prove that this selection and parameter binding can be interchanged with every other construct and this process is repeated for “left” to “right” until a selection is found.

\[ \text{reduce}(\#, \Delta(l \leftarrow L)) \]  
*Theorem V15* proves interchange with parameter binding;

*Theorem V14* proves interchange with selection;

\[ \#^{e}(ve_1, ..., ve_n) \]  
*Theorem V11* proves interchange with parameter binding;

*Theorem V12* proves interchange with selection;

\[ \Delta(r \leftarrow R) \]  
This parameter binding can be composed with \( \Delta(j \leftarrow J) \) yielding \( \Delta((j,r) \leftarrow J \times R) \), interchange with the selection is not necessary, because parameter binding \( \Delta(j \leftarrow J) \) “absorbs” \( \Delta(r \leftarrow R) \);
\( \Delta(r \leftarrow R(j)) \) This parameter binding can be composed with \( \Delta(j \leftarrow J) \) yielding \( \Delta((j,r) \leftarrow J \times R(j)) \), the search must now continue for the selection;

\( [s(f)] \) 

Goal reached, stop induction

If the parameterized parameter binding has been encountered Theorem 10b must be applied, otherwise Theorem 10a.

\( \square \)

In the sequel, we will refer to the generalised contraction as contraction and we will not distinguish it from the contraction defined in Theorem V10.

**Reference Function**

In the next section, we need a function to determine which data elements of data structures are referenced in a certain view expression. This function \( \text{Ref} \) returns a list of referenced view expressions.

**Definition 6.22**

Let \( \text{Ref} : \text{Pow(ViewExpression)} \rightarrow \text{ViewExpressionList} \), then the \( \text{Ref} \) for the basic elements of \( V\text{-cal} \) yields:

\[
\text{Ref}(\land(b, R, dp, ip) ve) = \bigcup_{v \in \text{Ref}(ve)} \land(b, R, dp, ip) v;
\]

\[
\text{Ref(DataStructure)} = \text{DataStructure};
\]

\[
\text{Ref}(\#(ve_1, ve_2, \ldots, ve_n)) = \bigcup_{k \in \{1,2,\ldots,n\}} \text{Ref}(ve_k);
\]

\[
\text{Ref}(\pi_{x_1}(ve), \ldots, \pi_{x_n}(ve)) = \bigcup_{k \in \{1,2,\ldots,n\}} \text{Ref}(\pi_{x_k}(ve))
\]

\( \square \)

Note that from the above basic definitions, we can derive the \( \text{Ref} \) function for the other elements as well, by using definitions V1 to V5, e.g.

\[
\text{Ref}(\{i\} ve) = \text{Ref}(\lor((i:i), True, id, id) ve) = \bigcup_{v \in \text{Ref}(ve)} \lor((i:i), True, id, id) v
\]

**Example 6.6**

The reference function returns the elements of data structures actually involved in the computation. For example consider the following program fragment:

\[
\text{Ref}(\Delta(i \leftarrow I) \ \text{reduce}(*, \Delta(k \leftarrow K) ([2^*i, k]A + [i, k]B))) =
\]

\[
(\Delta(i \leftarrow I) \Delta(k \leftarrow K) [2^*i, k]A, \Delta(i \leftarrow I) \Delta(k \leftarrow K) [i, k]B)
\]

The reference list of this example contains two view expressions, which describe the elements that are accessed in the program fragment. Elements that occur twice are eliminated by the union operator:
\[ \text{Ref} (\Delta(i \leftarrow (0:10)) [i]A) +\dagger (\Delta(i \leftarrow (0:10)) [i + 4]A) = \\
\{ \Delta(i \leftarrow (0:10)) [i]A \cup \Delta(i \leftarrow (0:10)) [i + 4]A \} = \Delta(i \leftarrow (0:14)) [i]A \]

It should be noted that the union operator cannot resolve every expression at “compile-time” and needs to postpone parts to “run-time”.

Normal Form
The view calculus V-cal has a normal form, which is obtained as follows:

1. First all *views* are composed and after composition all views are expanded to parameter bindings and selections;
2. All superscalar functions and reduces with superscalar functions are expanded to their scalar form by interaction with selections and parameter bindings;
3. All possible (generalised) contractions on parameter bindings and selections are applied;

This normal form is obtained by the application of a finite sequence of contractions and other rewrite rules, because the composition of views and the contraction of parameter bindings and selections always decreases the number of elements in the view expression with one. All other reductions are invariant to the number of elements introduced.

Having described the rewrite rules for the view expression part of V-cal, we continue with the rules for clause reductions and the interactions between the two.

6.4. Clauses

View expressions in V-cal describe the functional or state-less part of an algorithm, whereas clauses describe the state-to-state semantics. We will not go into the details of clause-semantics, but refer to the work of Loeckx and Sieber [Loeckx87]. Here, they specify the operational semantics of a while-structured language, that matches that of V-cal. The introduction of the semantics of clauses is kept rather informal and only those elements needed in the sequel of this chapter are defined.

We distinguish between the following types of clauses: *view definitions, assignments, conditionals, and repeat* constructs. In manipulating clauses, we focus on the most optimal ordering of computations within *(intra)* and between *(inter)* clauses. We therefore discuss the syntax of clauses including the definition of intra- and inter-clause orderings in Section 6.4.1. Section 6.4.2 presents intra-clause optimisations and the last section discusses inter-clause dependencies.
6.4.1. Definition of Clauses

A *clause* is a construct that incorporates one or more view expressions or clauses and defines a state-to-state transformation. This in contrast to a view expression that defines a state-to-value transformation. In terms of abstract syntax, a *Clause* is one of the following constructs:

\[\text{Clause} = \text{ViewDefinition} | \text{Assignment} | \text{Conditional} | \text{Repeat}\]

**Orderings**

As for other state-based languages, dependence analysis results in a data- and control-dependence graph. Much research involving data and control-dependence graphs, in particular where representation and efficient implementation is concerned, is reported in literature [Wolfe89, Zima91]. There is little to add to that, except for the exploitation of the multi-dimensional character of the assignments and conditions of V-cal. We will therefore focus on the representation and the manipulation of the data dependence graph for these parameter bindings. Data dependencies within assignments are expressed in terms of orderings on the parameter binding.

**Definition 6.23 Parameter Binding Orderings**

An ordering operator \(\diamond\) for parameter bindings defines a partial ordering on \(N^b\), where \(\diamond\) is defined as:

\(\diamond_I = \{ i, j \in I \mid \text{Rel}(i, j) \}\)

where \(\text{Rel}: N^d \times N^d \rightarrow T\) is true when a relation between \(i\) and \(j\) exists.

In addition, we consider the following two standard orderings for an index set \(I\):

- \(\leftrightarrow\) \(\iff\{ i, j \in I \mid i \neq j \land i < j \}\)
- \(\lnot\) \(\iff\{ i, j \in I \mid \text{True} \}\), i.e. there is no partial ordering

**ViewDefinition**

A *view definition* (re-)defines view identifier(s) by assigning view expression(s). The view definition adds or replaces aliases in the state-space without changing any values in data structures.

\[\text{ViewDefinition} = \Delta(k \leftarrow K) \{ [f(k)]\text{ViewIdentifier} = \text{ViewExpression}(k) \}\]

**Assignment**

An assignment initializes or changes the values associated with (a part of) a data structure. The assignment in effect consists of a pair of view expressions,
where the view expression to which the other view expression is assigned may not contain any functions or operators. The syntax for the assignment is as follows:

\[
Assignment = \Delta(k \leftarrow K) \odot ([f(k)]ViewIdentifier := ViewExpression(k))
\]

The view-expression on the left is identified with Left and the one on the right with Right. For example, for a statement \(S\) the expression \(S.Left = \Delta(k \leftarrow K) \odot [f(k)]ViewIdentifier\) and \(S.Right = \Delta(k \leftarrow K) \odot ViewExpression(k)\)

**Control Statements**

Similar to Booster, \(V-cal\) knows two control statements: the conditional and repeat-statement. The conditional as well as the repeat-statements have a view expression yielding a Boolean value as argument. Depending on the value of this expression the flow of control changes. The template is specified below, where Body, ThenPart, and ElsePart consist of lists of clauses.

\[
\text{If } \text{Cond } \text{Then } \text{ThenPart } \text{Else } \text{ElsePart } \text{End}
\]

\[
\text{While } \text{Cond } \text{Do } \text{Body } \text{End}
\]

**6.4.2 Intra-Clause Reductions**

Intra-clause reductions manipulate the denotation of the parameter bindings and orderings. These reductions involve the reordering of assignments, view definitions etc.

**View Expressions**

The computation of view expressions involves the combination and concatenation of various intermediate results. For example, the view expression on the left can be seen as two computations, namely, the computation of \(T\) and the summation of \(T\) and \(C\):

\[
C + (A*B) \Rightarrow T = A*B \text{ and } C + T
\]

This can formalised in \(V-cal\) with the help of intermediate data structures.
View Definitions

In a view definition, a parameter binding that occurs on both sides of the equivalence sign can be extracted from the view definition. The view definition is thus reformulated from a "vector" of definitions to a "vector" definition.

Definition C1 View Definitions

\[ \Delta(k \leftarrow K) \{ [f(k)]V = ve(k) \} \Leftrightarrow \{ \Delta(k \leftarrow K) [f(k)]V = \Delta(k \leftarrow K) ve(k) \} \]

Assignments

The same equivalence as defined above for a view definition holds for an assignment.

Definition C2 Assignments

\[ \Delta(k \leftarrow K) \{ [f(k)]V := ve(k) \} \Leftrightarrow \{ \Delta(k \leftarrow K) [f(k)]V := \Delta(k \leftarrow K) ve(k) \} \]

If the assignment symbol ':=' and the view definition equivalence '=' are considered as superscalar binary functions, the first definition can be compared with the parameter binding extraction for functions as defined in Theorem V12. Therefore, the parallel ordering for the parameter binding of i is consistent with intuition, because it implements the element-wise semantics of the assignment.

Parameter Binding Decompositions

Parameter bindings can be decomposed into two parameter bindings and corresponding orderings:

Definition C3 Parameter Binding Decompositions

Let \( X = I \times J \) and \( \Diamond = \{ R(i) \land Q(i,j) \} \), then

\[ \Delta(x \leftarrow X) \Diamond \text{Clause}(x) \Leftrightarrow \Delta(i \leftarrow I) \Diamond(R(i)) \Delta(j \leftarrow J(i)) \Diamond(Q(i,j)) \text{Clause}(j) \]

The interchange of parameter bindings with parallel orderings is permitted:

Definition C4 Parameter Binding Interchange

\[ \Delta(i \leftarrow (a,P)) \parallel \Delta(j \leftarrow (b,R(i)))) \parallel \text{Clause}(i,j) \Leftrightarrow \Delta(j \leftarrow (b,\text{True})) \parallel \Delta(i \leftarrow (a,P \land R(i)))) \parallel \text{Clause}(i,j) \]
6.4 CLAUSES

6.4.3 Inter-Clause Orderings and Reductions

Apart from orderings within clauses, we also consider orderings between (sets of) clauses. To determine the ordering we need a number of functions on clauses to extract certain properties. These properties include the use of data in clauses and which data is actually defined or used. Data is grouped in data structures and data structures are identified by variables. The set of all possible variables is referred to as Variable.

Parts of a data structure used or changed in operations are described by use of view expressions. This implies that for one data structure various parts may be used or changed. The set of all possible view expressions is referred to as Structure.

\[
\text{Def, Ref} \quad : \quad \text{Clause} \times \text{Variable} \to \text{Structure}
\]

The functions Def and Use are defined for each of the clauses as follows, where the Ref function for view expressions was defined in the previous section.

\[
\text{Def}(\text{Assignment}) = \text{Def}(\Delta(i \leftarrow I) \odot \text{Assignment}(i)) = \Delta(i \leftarrow I) \odot \text{Ref}\left(\text{Left}(i)\right)
\]
\[
\text{Ref}(\text{Assignment}) = \text{Ref}(\Delta(i \leftarrow I) \odot \text{Assignment}(i)) = \Delta(i \leftarrow I) \odot \text{Ref}\left(\text{Right}(i)\right)
\]

\[
\text{Def}(\text{If-clause}) = \text{Def}(\text{ThenPart}) \cup \text{Def}(\text{ElsePart})
\]
\[
\text{Ref}(\text{If-clause}) = \text{Ref}(\text{ThenPart}) \cup \text{Ref}(\text{ElsePart}) \cup \text{Ref}(\text{Cond})
\]

\[
\text{Def}(\text{While-clause}) = \text{Def}(\text{Body})
\]
\[
\text{Ref}(\text{While-clause}) = \text{Ref}(\text{Body}) \cup \text{Ref}(\text{Cond})
\]

For data dependence testing, we introduce the following notions of Change and Use view expressions of data structures. The Change view expression of a data structure \(A\) references a (subset) of \(A\)'s index set that is changed or modified in a certain clause. The Use view expression of a data structure references the subset that is used or accessed in a certain clause. The Change or Use view expressions are derived as follows:

**Definition 6.24**
Consider the following parameter binding defined on \(A\) with index set \((a,P)\):

\[
\text{Ref}(A) = \Delta(k \leftarrow I) \odot [ip(k)]A \\
\text{Def}(A) = \Delta(k \leftarrow J) \odot [ip'(k)]A
\]

then \(Use(A) = \{ ip(i) \mid i \in I \}\) \( \text{and} \) \( Change(A) = \{ ip'(i) \mid i \in J \}\)

\(\square\)

**Example 6.7**
The definition of Change (or Use) sets is illustrated by the following view function:
Def(A) = \Delta((i_1,i_2) \leftarrow (2:4 \times 3:5)) [2^i_1,i_2 + 1]A

The Change (or Use) set is:

Change(A) = \{ (2^i_1,i_2 + 1) | (i_1,i_2) \in (2:4 \times 3:5) \}

It is clear that in some cases this representation can be simplified. For example, when the ip function is linear the Change set can be simplified:

Change(A) = \{ (2^i_1,i_2) | (i_1,i_2) \in (2:4 \times 4:6) \}

Ordering between Clauses

The ordering between clauses is defined in terms of the Change and Use sets. Here we distinguish between various types of dependencies, such as data, output, and anti dependencies [Banerjee88a + b].

**Definition 6.25**

Let $C_1$ and $C_2$ be clauses between which a lexicographic order $C_1 > C_2$ exists, then the following dependencies exist if the condition is true:

*Data Dependence:* \( \text{Change}(C_1,X) \cap \text{Use}(C_2,X) \neq \emptyset \)

*Anti dependency:* \( \text{Use}(C_1,X) \cap \text{Change}(C_2,X) \neq \emptyset \)

*Output dependency:* \( \text{Change}(C_1,X) \cap \text{Change}(C_2,X) \neq \emptyset \)

If there is a dependence, this will be denoted as: \( \text{Dependence}(C_1, C_2, X) \). 

\[ \square \]

The derivation of these dependencies is a difficult problem, but often simplifications are possible. These optimisations are out of the scope of this thesis.

**Definition 6.26**

Let $C_1$ and $C_2$ be clauses between which some dependence exists, the ordering between these clauses is then defined as follows:

\[ \text{Vars} = \{ X \in C_1 \text{ and } X \in C_2 | \text{Dependence}(C_1, C_2, X) \} \]

For each $X$ in Vars, a pair of References or Definitions to $R_1 \in \text{Def}(X,C_1)$ and $R_2 \in \text{Ref}(X,C_2)$ is selected:

\[ R_1 = \Delta(i \leftarrow \lambda) \circ_1 [f(i)]X \text{ and } R_2 = \Delta(j \leftarrow J) \circ_2 [g(j)]X \]

then the partial ordering $R_1 \Phi R_2 = \{ i, j \in I \times J | \text{Rel}(i,j) \}$ where $\text{Rel}: I \times J \rightarrow T$ is true when a relation between $i$ and $j$ exists. This relation is defined as follows, where $\circ_1'$ is the ordering of $I$ after application of $f$ and $\circ_2'$ is the ordering of $J$ after application of $g$: 
∀x ∈ I ∀y ∈ J ( f(x) ⊤_1 f(y) ⇔ x ⊤_1 y )
∀x ∈ I ∀y ∈ J ( g(x) ⊤_2 g(y) ⇔ x ⊤_2 y )
Φ = \{ i, j ∈ I × J \mid \exists x ∈ I \exists y ∈ J \{ f(i) ⊤_1 f(x) ∧ g(y) ⊤_2 g(j) ∧ f(x) = g(y) \} \}

The ordering between the Clauses C_1 and C_2 is defined as follows:

C_1 ∩ C_2 = \bigcup_{X \in \text{Vars}} \bigcup_{RI \in \text{Ref}(X,C_1)} \bigcup_{R2 \in \text{Ref}(X,C_2)} R_1 ∩ R_2

In summary, the ordering between clauses is defined through the union of relations between each reference to a variable of the two clauses. The above definition links the intra-clause orderings with the inter-clause orderings. The advantage of the formalisation of dependencies in terms of relations on parameters is threefold. First, they allow for easy manipulation of parameters with reduction-rules. Second, the data dependence information is defined within the formal framework and not separate. Third, only the necessary elements of the data-dependence structure are represented. Below, we give a number of examples of the derivation of inter-clause orderings.

Denotation

For convenience, we use the following standard orderings for a domain set I × J:

; ⇔ \{ (i, j) ∈ I × J \mid i ≠ j ∧ i < j \}
\| ⇔ \{ (i, j) ∈ I × J \mid \text{True} \}, i.e. there is no relation

Example 6.8

A predicate with parameters of both assignments denotes the ordering. An example is:

C_1 ∩ C_2 = Δ(i ← I) ∗ \{ [i]B := [i−1]B ∗ [i−2]B \} ∩ Δ(j ← I) ∩ \{ [j]C := [j]B ∗ 2 \}

The data-dependence between elements of B determines the ordering between the above assignments. By intersecting the Change and Use sets the dependence becomes apparent.

\text{Change}(C_1,B) \cap \text{Use}(C_2,B) = \text{Index}(Δ(i ← I) [i]B) \cap \text{Index}(Δ(j ← I) [j]B) = I

The Def and Ref functions for C_1 and C_2, respectively, return the following two view expressions:

\text{Def}(C_1,B) = Δ(i ← I) ∗ [i]B and \text{Ref}(C_2,B) = Δ(j ← I) \cap [j]B

The ordering between the assignments is determined by the modified orderings of the Def and Ref view expressions. This is the original ordering of Def and
Ref after application of the index propagation functions. In this particular case, however, both functions are equal to the identity function and the modified orderings are equal to their originals. This results in:

\[ \Phi = \{(i, j) \in I \times J \mid \exists x \in I \exists y \in J \ i < x \land x = j \} = \{(i, j) \in I \times J \mid i < j \} \]

The relation is thus expressed through the parameters \(i\) and \(j\) and equals \(i < j\). Intuitively, this corresponds with the fact that the value for \([i]B\) must have been written before \([j]B\) can be read. We denote the ordering between clauses as follows:

\[ \Delta(i \leftarrow I) \ast \{(i)B := [i-1]B \ast [i-2]B\} \ \Phi(i < j) \ \Delta(j \leftarrow I) \ast \{(j)C := [j]B \ast 2\} \]

The advantage of this representation is the use of the same parameters in both clauses as well as in the ordering specification. A change in the parameters directly operates on the ordering specification.

**Inter-Clause Reductions**

Inter-clause reductions define transformations that rewrite the internal ordering of clauses to a more optimal form. We will return to the criteria for the most optimal ordering in Chapter 7. Given a certain ordering for clauses, we can define inter-clause reductions to optimise overall program performance. An example of such a reduction is the extraction of a parameter binding from two clauses:

**Definition C5 Clause fusion**

\[ \Delta(i \leftarrow I) \parallel C_1(i) \parallel \Delta(i \leftarrow I) \parallel C_2(i) \iff \Delta(i \leftarrow I) \parallel \{ C_1(i) \parallel C_2(i) \} \]

Another example of an inter-clause reduction is based on an example from the previous section:

\[ \Delta(i \leftarrow I) \ast \{(i)B := [i-1]B \ast [i-2]B\} \ \Phi(i < j) \ \Delta(j \leftarrow I) \parallel \{(j)C := [j]B \ast 2\} \iff \Delta(i \leftarrow I) \ast \{(i)B := [i-1]B \ast [i-2]B ; (i)C := [i]B \ast 2\} \]

We will not elaborate any further on inter-clause reductions, because these are out of the scope of this thesis. Moreover, [Zima91] and [Wolfe89] describe many of the required optimisations albeit in a different formalism than \(V-cal\). These optimisations include transformations like loop-fusion and loop-splitting.
Chapter 7
Semantics and Translation of Booster

Having defined V-cal in the previous chapter, we can describe the semantics of Booster and its annotation modules in terms of this intermediate formalism. Whilst describing the semantics of Booster, we will surpass the issues of parsing and error-checking and focus on the translation of complete syntactic entities at a time. We will also elaborate on the machine independent optimisation of translated Booster programs on the V-cal level.

7.1 Translation of Booster to V-cal

The translation of Booster to V-cal is discussed by translating each of the constructs that constitute Booster: simple views, view statements, data fields, view functions, content and control statements, and finally functions and modules.

7.1.1 Views

In Booster we have distinguished three kinds of view definitions: views (Section 4.2.4), view statements (Section 4.3.1 - 4.3.3), and view functions (Section 4.3.4). Each of these views is translated into the triple view representation \((K, dp, ip)\) of V-cal. We will start with the semantics of the views, followed by the semantics of view statements, and finally the semantics of view functions.

Views
For views, we use the following conventions:

- \(expr\) is an integer-expression that contains parameters, constants, and/or integer-operators;
- \(I\) is an index set with bounding vector \(x\) and predicate \(P\).
- \(X = (a, Q)\) and \(Y = (b, R)\).
- \(A\) and \(B\) are shapes.

For each of the possible views the semantics are defined as follows:

Selection of a single element:

\[ expr \quad I = ((expr:expr), True). \]
Selection of a range of elements:
\[ \text{expr1: expr2} \enspace I \equiv ((\text{expr1: expr2}), \text{True}). \]

Intersection of index sets:
\[ X \cap Y \enspace I \equiv (a \land b, Q \land R). \]

Union of index sets:
\[ X \cup Y \enspace I \equiv ((\text{min}(X \cup Y), \text{max}(X \cup Y)), \{ i \mid i \in X \lor i \in Y \}). \]

The not operator on an index set
\[ (\neg) \enspace I \equiv (a, \{ i \mid i \neq b \} \land Q). \]

Selection of all elements in a dimension:
\[ (\_\_\_) \enspace I \equiv X. \]

Content selection:
\[ [\text{A relop B}] \enspace I \equiv (y, Q \land \{ i \mid i \in A \land \text{relop B} \}), \text{where (y, Q) = index(A)}. \]

Content function:
\[ \{ f(A) \} \enspace I \equiv ((0: \text{size}(fA)) - 1, \{ i \mid i \in \text{index}(fA) \}). \]

Multi-dimensional views
Each of the above view forms can occur in a multi-dimensional selection of which the semantics are defined as follows:

\[ [x_1, x_2, \ldots, x_n] \text{ where view } V_i = (K_i, dp_i, ip_i) \text{ defines the semantics of the view } X_i, \text{ then } V = ((K_1 \times K_2 \times \ldots \times K_n), (dp_1, dp_2, \ldots, dp_n), (ip_1, ip_2, \ldots, ip_n)). \]

Composition of Views
The semantics of the view composition is expressed as a view composition of the individually translated views:

\[ A[X][Y] \equiv Y(X(A)) \]

For example, consider the following view concatenation:

\[ A[1:4][2:4], \text{ translates to } V = (1:4, \text{True, id, id}), \quad W = (2:4, \text{True, id, id}) \]
\[ V \circ W = (2:4 \land 1:4, \text{True, id, id}) = (2:4, \text{True, id, id}) \]

View Statements
The translation of view statements is specified in terms of the basic operators from which they are constructed. In Chapter 4, however, a distinction was made between the composition of "simple" views and the composition of views.
through view statements. In "simple" views, the composition equals the intersection of the index sets, whereas in view statements the left part is renumbered before being assigned to view identifiers. To facilitate the description of the semantics, we introduce two views that implement the realignment and the renumbering of index sets, respectively. Realignment yields an index set that is numbered from zero onwards and renumbering always yields index sets in which all elements are consecutive and numbered from zero onwards. Note that the renumber operation is only valid for 1-dimensional structures.

**Definition 7.1**

The views realign : \( N^d \rightarrow N^d \) and renumber : \( N \rightarrow N \) are defined as follows:

\[
\text{realign} = ((0,\infty), \text{True}, dp, ip),
\]

where \( dp((l, u)) = (0, u - l), \ ip(i) = i + l; \)

\[
\text{renumber} = ((0,\infty), \text{True}, dp, ip),
\]

where \( dp((b, R)) = (0, \text{size}((b, R)) - 1), \ ip(I_i) = pos(i); \)

The \( dp \) functions of both views do not unconditionally satisfy the requirement \( dp(I \cap J) = dp(I) \& dp(J) \), (see Definition 6.16). For example, if we consider the \( dp \) function of realign then:

\[
dp(3:6 \cap 5:10) = dp(5:6) = 0:1 \neq dp(3:6) \& dp(5:10) = 0:2 \& 0:5 = 0:2
\]

Fortunately, in Booster, it always holds that \( I \subseteq J \) or \( J \subseteq I \), because selections always have to be within bounds. The proof is given below:

**Theorem 7.1**

Let \( I \) and \( J \) be index sets for which \( I \subseteq J \), then if \( dp((b, R)) = (0, \text{size}((b, R)) - 1) \) the following property holds:

\[
dp(I \cap J) = dp(I) \& dp(J)
\]

**Proof:**

Because \( I \) defines an index set that is a subset of the one defined by \( J \), the following holds:

\[
I \cap J = I \Rightarrow dp(I \cap J) = dp(I)
\]

Furthermore, because \( I \subseteq J, I = (a, P) \), and \( J = (b, R) \) it holds that \( b_l \leq a_l \land a_u \leq b_u \)

\[
\Rightarrow a_u - a_l \leq b_u - b_l, \text{ hence}
\]

\[
dp(I) \& dp(J) = dp(I)
\]

This proves the property \( dp(I \cap J) = dp(I) \& dp(J) \).

\( \Box \)
Note that the domain propagation function for the realign view \( dp((I, u)) = (0, u-l) \) is just a special case of the domain propagation function of the renumber view. The above theorem therefore also holds for this case.

**Semantics of view statements**

The above functions are used in the description of the semantics of view statements, of which the general templates for Booster are given below, with their equivalents in V-cal. We have split the general view statement in four different categories:

- view statements without free variables;
- view statements with free variables, but without domain specifications;
- view statements with free variables, but a single domain specification;
- view statements with free variables with two domain specifications.

**View statement without free-variables**

Booster allows for the definition of view statements without free variables, but with a (composition of) views. The semantics for this construct are as follows:

\[
T \leftarrow A[I_1, I_2, \ldots, I_n] \quad \text{if } I_i = (x_i, P_i), \text{ where } \text{card}(I_i) = 1, \text{ then } \\
\pi_i(T) = \text{renumber}(I_i)
\]

\[
T \leftarrow A[V_1, V_2, \ldots, V_n] \quad \text{if } V_i = (K_i, dp_i, ip_i), \text{ then } \\
\pi_i(T) = \text{renumber}(V_i(\pi_i(\text{Index}(A))))
\]

Examples are, where \( V_{even} \) is the translation of the function \( \text{even} \):

\[
T \leftarrow A[2 | 4]; \quad T = \text{renumber}([2|4]A) \\
T \leftarrow B[2:4, \text{even}(2)]; \quad T = (\text{renumber}([2:4]\pi_j(B)), \text{renumber}((\text{Even}(\pi_j(B)), 2)))
\]

**View statement with free-variables, but without domain-specifications**

A constraint on this type of view statement is that the function \( f \) must be a monotonic increasing or decreasing, integer function for which an analytical inverse integer \( f^{-1} \) exists and which can be derived by the compiler. The general view statement below defines the view \( V \), where the index of \( A \) equals \((x_1, x_u, P)\).

\[
T[i_1, i_2, \ldots, i_d] \leftarrow A[f(i_1, i_2, \ldots, i_d)] \\
V = (K, dp, ip), \text{ where } K = ((0, \infty), \text{True}), dp = f^1, ip = f
\]

An example is:

\[
V[i] \leftarrow A[4*i + 1]; \quad \text{ip}(i) = 4*i + 1, \text{ and } \text{ip}^{-1}(i) = (i - 1) \div 4
\]
View statement with a single domain-specification:
A constraint for this view statement is that the function \( f \) must be an (integer) function defined on the domain \((0.g(n))\).

\[
T \{ g(n_1, n_2, \ldots, n_m) \ [i_1, i_2, \ldots, i_d] \rightarrow A[f(i_1, i_2, \ldots, i_d)] \}
V = (K, dp, ip), \text{ where } K = ((0.g(n)), True), \text{ } dp = id, \text{ } ip = f
\]

An example is:

\[
V \{ 3 \div 4 \} \ [i, j] \rightarrow A[i + j];
V = (K, dp, ip), \text{ where } K = (0:2 \times 0:4), \text{ } dp = id, \text{ } ip = i_1 + i_2.
\]

View statement with a double domain-specification:
A constraint on this view statement is that the function \( f \) must be an (integer) function defined on the domain \((0.g_p(n))\).

\[
T \{ g(n_1, \ldots, n_m, p_1, \ldots, p_m) \ [i_1, \ldots, i_d] \rightarrow
A \{ (n_1, \ldots, n_m) \ [f(i_1, \ldots, i_d)] \}
V = (K, dp, ip), \text{ where } K = ((0, \infty), True), \text{ } dp = g_p, \text{ } ip = f.
\]

An example is:

\[
V \{ n \div 2 \div m \div 3 \} \ [i, j] \rightarrow C \{ n \div m \} \ [2*i, 3*i];
V = (K, dp, ip), \text{ where } K = ((0, \infty), True), \text{ }
dp(i) = (i_1 \div 2, i_2 \div 3), \text{ } ip(i) = (2*i_1, 3*i_2).
\]

Data Fields
Data fields, introduced in Section 4.4.4, have semantics that differ from those of the view statements discussed above. They do not define a relation between index sets of the left- and right-hand side of a statement, but relate a view identifier to a data set and corresponding index set which does not refer to any shape, e.g.:

\[
V \{ 20 \div 30 \} \ [i, j] \rightarrow \text{i+j};
\]

Here a view identifier with an index set \(20 \times 30\) and associated data set is defined comprising of elements that equal the sum of the indices associated with that particular data element. The semantics for these data fields are expressed as follows:

\[
T \{ g(n_1, n_2, \ldots, n_m) \ [i_1, i_2, \ldots, i_d] \rightarrow \text{Expr}(i_1, i_2, \ldots, i_d)
\]

\[
A(i \rightarrow (0.g(n_1,n_2,\ldots,n_m))) \{ T[i_1,i_2,\ldots,i_d] = \text{Expr}(i_1,i_2,\ldots,i_d) \}
\]
View Functions

A view function in Booster encapsulates a number of view statements. The semantics of the view function dictates that actual arguments, when passed to the view function, have to be renumbered and realigned. Apart from the interface notion the semantics of view functions are described as a composition of the individual view statements.

Definition of view functions

In the definition of view functions, a number of view statements relates the input and output arguments of the function. This scheme is specified as follows:

```
VIEW FUNCTION name ( I_1, I_2, ..., I_n ) -> ( O_1, O_2, ..., O_m );
BEGIN
    relations
END;
```

In terms of V-cal this becomes:

\[ O_1 = V_1(I_{x_1}); \]

\[ \vdots \]

\[ O_m = V_m(I_{x_m}); \]

Note that the input arguments may either be shapes, or parameters.

Application of view functions

The application of view functions in view and content-statements involves the substitution of the body of the view function in the program replacing the formal arguments by the actual arguments. For the definition of the semantics, we will make a distinction between view function calls that occur in view concatenations in a view expression and those that do not.

View function calls in view statements

A call to a view function \( X \) with actual arguments \( A_1, A_2, ..., A_n \) is in this case translated according to the following template.

\( (B_1, B_2, ..., B_n) \leftarrow X(A_1, A_2, ..., A_n) \)

Depending on the dimension of the index set, the view function call is translated to a sequence of operations. First, the actual input parameters are assigned to the formal input parameters, then the body of the view function is "executed", and finally the output formal parameters are assigned to the actual output parameters. The template for this operation is:
7.1 TRANSLATION OF BOOSTER TO V-CAL

\[ I_x = \text{realign}(A_x); \quad x \in 1{:}n \]

\[ \text{Body}(X); \quad x \in 1{:}n \]

Example 7.1

An example of this is the view function \texttt{Unzip} from Section 4.3.4, which is translated to \textit{V-cal} as follows

\[
(\text{Even}, \text{Odd}) = \text{Unzip}(Q): \\
\text{Even} = \sqrt{(dp(i) = (i+1) \div 2 + (i+1) \mod 2, ip(i) = 2*i)Q} \\
\text{Odd} = \sqrt{(dp(i) = (i+1) \div 2, ip(i) = 2*i + 1)Q}
\]

The call to the function \textit{Unzip} equals:

\[
(\text{Even}_\text{elements}, \text{Odd}_\text{elements}) <- \text{Unzip}([2{:}6]A)
\]

Application of the view function \texttt{Unzip} to a viewed shape \( A \) \((0{:}n)\) involves realignment and renumbering of the actual argument \( \sqrt{(b = (2,6))A} \). (Because \( 2{:}6 \) is a consecutive set there is no need for a renumbering.)

\[
I = \text{renumber}(\sqrt{(b = (2,6), ip(i) = i)A}) = \sqrt{(b = (0,4), ip(i) = i+2)A} = \\
\text{realign}(\sqrt{(b = (2,6), ip(i) = i)A}) = \sqrt{(b = (0,4), ip(i) = i+2)A}
\]

Next the realigned actual argument is substituted in the body of the view function for the formal argument \( Q \):

\[
\text{Even}_\text{elements} = \\
\sqrt{(dp(i) = (i+1) \div 2 + (i+1) \mod 2, ip(i) = 2*i) \sqrt{(b = (0,4), ip(i) = i+2)A}} = \\
\sqrt{(b = (0,2), ip(i) = 2*i+2)A}
\]

\[
\text{Odd}_\text{elements} = \\
\sqrt{(dp(i) = (i+1) \div 2, ip(i) = 2*i + 1) \sqrt{(b = (0,4), ip(i) = i+2)A}} = \\
\sqrt{(b = (0,1), ip(i) = 2*i+3)A}
\]

As a result the view function call is eliminated and we have obtained a direct expression for \textit{Even\textsubscript{elements}} and \textit{Odd\textsubscript{elements}} in terms of \( A \).

7.1.2 Content Statements

The translation of content statements involves the transformation to assignments in conjunction with the translation of the views that are used in content statements. Of main interest for this translation is the order in which the multi-dimensional assignments are traversed, which depends on intra-content-statement dependencies.
General translation scheme

The translation of content statements consists of the denotation of each view and view function in terms of the triple view representation \( \sqrt{(K, dp, ip)} \). The next step is the resolution of all view compositions and the normalisation of view expressions. For each free variables that occurs in the content statement a range or index set must be derived. This index set is bound to the parameter in a parameter binding with the assignment as an argument.

The composition of all views and normal form was already discussed in the previous chapter. We therefore focus on determining the ranges for the free variables. These are derived using the \( \text{Def} \) and \( \text{Ref} \) expressions of the assignment, e.g.

\[
\begin{align*}
\text{Def}(S) &= \sqrt{(K_i, id, f_i)}A \\
\text{Ref}(S) &= \sqrt{(J_i, id, g_i)}B
\end{align*}
\]

For this parameter \( i \) the following restrictions hold:

\[
\forall j \in K_i f_i(j) \in \text{Index}(A) \land \forall j \in J_i g_i(j) \in \text{Index}(B)
\]

The solution of these equations in \( i \) yields the desired range for \( i \). This is illustrated with an example.

Example 7.2
Consider the following program fragment:

\[
\begin{align*}
\text{SHAPE A, B, C (n) OF INT;} \\
A[2:4][2*i] &:= B[3*i] \ast C[i];
\end{align*}
\]

The \( \text{Def} \) and \( \text{Ref} \) functions yield the following views after composition, from which the conditions are derived:

\[
\begin{align*}
\text{Def}: & \quad \sqrt{(2:4, 2*i \in 2:4)} A \quad \Rightarrow 2*i \in 2:4 \\
\text{Ref}: & \quad \sqrt{(0:n-1, 3*i \in 0:n-1)} B \quad \Rightarrow 3*i \in 0:n \\
& \quad \sqrt{(0:n-1, 3*i \in 0:n-1)} C \quad \Rightarrow i \in 0:n
\end{align*}
\]

The range for the free variable \( i \) is deduced from the resulting constraints: \( 2*i \in 2:4 \land 3*i \in 0:n \land i \in 0:n \). As a consequence, \( i \in 1:2 \) and the following parameter binding is added, resulting in the translated assignment:

\[
\Delta(i \leftarrow 1:2) \parallel ([2*i] A := [3*i] B) \ast [i] C)
\]

Intra-content Statement Dependencies
A special case in the translation of content statements to \( V-cal \) is a statement with an internal dependency. An example is the following \( \text{Booster} \)-program fragment:
SHAPE A (n) OF INT;
A[0] := 1;
A[1:upb][i] := A[i-1]*2;

Here the computation of \( A[i] \) depends on the value of \( A[i-1] \) that results in an execution with a sequential ordering. This statement translates to the following V-cal expression:

\[
\Delta(i \leftarrow (1:n-1)) \bullet ([i]A := [i-1]A*2)
\]

with the sequential ordering operator '•'.

This is formalised using the Def and Ref functions introduced in Chapter 6. In Booster, there is an intra-dependence for statement \( S \) in variable \( X \) iff:

\[ \text{Change}(S,X) \cap \text{Use}(S,X) \neq \emptyset. \]

**Definition 7.2 Intra-statement dependencies**

Let the statement \( S \) be defined as:

\[
\Delta(i \leftarrow I) \odot \{ [f(i)]X := ve(i) \}
\]

For each \( R = \text{Ref}(X,ve(i)) \) we select a pair:

\[
D = [f(i)]X \text{ and } R = \Delta(k \leftarrow K) [g(i,k)]X
\]

then the partial ordering \( \odot(D,R) = \{ i, i' \in I \mid \text{Rel}(i,j) \} \), where \( \text{Rel}: I \times I \rightarrow T \) is true when a relation between \( i \) and \( i' \) exists. This relation is defined below, where the index set \( K'(i) \) contains those elements for which \( g(i,k) = f(j) \) for some \( j \in I \). The relation \( \text{Rel} \) of \( \odot(D,R) \) is defined as follows, where \( d = \text{dim}(I) \).

\[
\text{Rel}(i, i') = \{ i' \leq i + c \} \text{ iff a } c > 0 \text{ exists such that } c = \min( t \in N^d \mid \forall i \in I \forall i' \in I, i' \leq i + t \forall k \in K' f(i + t) \geq g(i',k))
\]

\[
\text{Rel}(i, i') = \{ i' \geq i - c \} \text{ iff a } c > 0 \text{ exists such that } c = \min( t \in N^d \mid \forall i \in I \forall i' \in I, i' \geq i - t \forall k \in K' f(i - t) \leq g(i',k))
\]

The ordering \( \odot \) of the statement \( S \) is equal to:

\[
\forall R = \text{Ref}(X,ve(i)) \odot(D,R)
\]

\[\square\]

**Sequentialization Criterion**

When the above dependence definition yields an ambiguous ordering of the form \( \{ j \geq i-c \land j \leq i+c \} \), the lexicographic order is chosen. This according to the sequentialization criterion in Section 4.4.3.
Example 7.3
Consider the Gauss-Seidel algorithm:


According to the definition the intra-statement dependence is equal to:

\[ \Diamond ((i-1 < i') \land (j-1 < j') \land (i' < i+1) \land (j' < j+1)) \]

This ordering is integrated in the assignment:

\[ \Delta(i, j) \leftarrow I \times J \setminus \{(i-1 < i') \land (j-1 < j') \land (i' < i+1) \land (j' < j+1)) \]


Another example involves a more complicated reference part:

\[ \Delta(i, j) \leftarrow \{2:10 \times J\} \setminus \{2*i, j\} A := reduce(*, \Delta(k \leftarrow \{0:1\}) [2*i+k, j] A \} \]

According to the definition the ordering is equal to \( \{ i' \leq i + c \} \), where

\[ c = \min(t \in N \mid \forall i \in I \forall i' \in I, i' \leq i + t \forall k \in K^f, f(i + t) \geq g(i', k) \} \]

Set \( K^f \) only has a single element 0, because \( 2*i \neq 2*i + 1 \) for all \( i \in I \)

\[ = \min(t \in N \mid \forall i \in \{2:10\} \forall i' \in \{2:10\}, i' \leq i + t \forall k \in \{0\} 2^*(i + t) \geq 2^*(i' + k) \] \]

\[ = \min(t \in N \mid \forall i \in \{2:10\} \forall i' \in \{2:10\}, i' \leq i + t 2^*(i + t) \geq 2^*(i') \} = 0 \]

Because \( c = 0 \), there is no dependence and the ordering is equal to the parallel one:

\[ \Delta(i, j) \leftarrow \{2:10 \times J\} \setminus \{2*i, j\} A := reduce(*, \Delta(k \leftarrow \{0:1\}) [2*i+k, j] A \} \]

7.1.3 Control Statements, Functions and Modules

The translation of control statements, functions, and modules into V-cal is less complicated than view and content statements. For the three control statements the translation is trivial, because equivalent constructs exists in V-cal.

Repeat Statements

The two repeat statements in Booster: \textit{iter} and \textit{while} directly translate to the equivalents in V-cal: the parameter binding and the \textit{while}-clause. The while-clause is translated to the following equivalent, where \( C \) is the translation of the Booster condition \( \text{cond} \) and \( B \) the translation of \text{body}.

\[
\text{WHILE} \ \text{cond} \ \text{DO} \ \text{body} \ \text{END} \quad \text{While} \ C \ {\{ B \}}
\]

The iter statement translates to a parameter binding with a ordering that depends on the inter-statement data dependencies:
7.1 TRANSLATION OF BOOSTER TO V-CAL

\[ \Delta(k \leftarrow (0:nr-1) \odot \{ B \}) \]

Conditional Statements
The conditional statement in Booster and in V-cal are each others equivalents.

\[ \text{IF cond THEN body1 ELSE body2} \quad \rightarrow \quad \text{If } C \text{ then } B1 \text{ else } B2 \]

Example 7.4
Consider the translation of a while-loop in Booster to V-cal:

\[
\begin{align*}
V & \leftarrow A \\
\text{WHILE } & \text{size}(V) > 0 \text{ DO} \\
V & \leftarrow V[1:upb]; \\
\text{END}
\end{align*}
\]

\[ \Delta(k \leftarrow K) \{ [k]V = [k]A \}; \quad \text{While } \text{size}(\text{Index}(V)) > 0 \{ V = \sqrt{(dp(i) = i-1, ip(i) = i+1)V} \} \]

Recursive view identifier definitions
The issue of the translation of recursive view definitions raised in the previous example, needs special attention. This type of view definitions is often used in Booster algorithms, for example in Gaussian elimination and Cholesky factorization. In general, the structure of a recursive view definition is as follows, where \( R \) is some view function:

\[
\begin{align*}
V & \leftarrow A; \\
\ldots \\
\text{ITER .. WHILE .. DO} \\
\ldots \\
V & \leftarrow R(V); \\
\ldots \\
\text{END}; \\
\ldots
\end{align*}
\]

The view identifier is defined on some base shape and within a repetition structure the view identifier is redefined in terms of itself (Note that the redefinition is not always directly in terms of the view identifier itself, but may involve a chain of view definitions which can be reduced.) The translation of these recursive view definitions involves two view assignments, where the second recursive view definition depends on a parameter \( k \).

\[
\begin{align*}
V & = A; \\
\ldots \\
\text{ITER .. WHILE .. \{} \\
\ldots \\
V & = R_k(A); \\
\ldots \\
\text{\}};
\end{align*}
\]

In translating and optimising the above program the goal is to find an expression for the view identifier \( V \) in terms of \( k \) and \( A \). This implies that the recur-
sive view definition needs to be resolved, which according to Theorem 6.20 can be achieved if the view \( R_k \) has the form \((b, \text{True}, dp, ip)\). We then obtain an expression of the form:

\[
V = (dp^{k-1}(b), \text{True}, dp^k, ip^k) A
\]

In those cases were the predicate of the views' bounding index set does not equal \text{True} or when the \( dp \) and \( ip \) functions are data-dependent, the view recursion cannot be completely resolved and has to be postponed to run-time.

### 7.2 Performance Enhancing Transformations

In the discussion on the translation of \textit{Booster} programs to \textit{V-cal}, we surpassed any notion of efficiency. In this section, we will therefore introduce metrics that define a relative performance measure for certain \textit{V-cal} constructs. Based on these metrics we can define performance enhancing transformations. The discussion on metrics and optimisations will be subdivided into index sets, functions, views, and machine specific optimisations.

#### 7.2.1 Index Set Related Optimisations

The choice to represent index sets as a set comprehension introduces two interesting performance issues. First, there are various possible set comprehensions that can be used to represent a certain set, some of which are more favourable than others from a performance point of view.

**Index Overhead Measure**

The translations of terms, expressions, and statements described in Section 7.1 show that for an index set \( I = (\alpha,P) \) potentially too much indices are generated \((N^\alpha)\), which have to be filtered out by the predicate function \( P \). Prior to the discussion on optimisations on index sets, we need to present some kind of \textit{cost} function, or \textit{overhead} measure. For example the set \((0, 1, 2)\) may be described by \((0:2, \text{True})\), or by \(((0:6), i \leq 2)\). We prefer the first description because this form enumerates all elements, without any need to test the predicate and is therefore less expensive from a computation point of view. This is formalised in the \textit{index set overhead measure}:

**Definition 7.5**

Let \( I = (\alpha,P) \), and \( IM: \text{IndexSet} \to N \), then \( IM(I) = \frac{\text{size}(N^\alpha)}{\text{size}(I)} \)

The function applied to our example index set representations yields: \( IM((0:2, \text{True})) = 1 \) and \( IM(((0:5), i \leq 2)) = 2 \). Hence, the criterium for index set optimisations for an index set \( I \) equals:
\[(b,R) = \min\{ IM((a,P)) \mid (a,P) = I \}\]

The index overhead measure can also be applied to views, and by definition also to parameter bindings.

### 7.2.2 View Composition Optimisations

An important element of the translation of Booster to V-cal and optimisations on V-cal programs are view compositions. In practice, however, not all view compositions can be resolved at compile-time, some view compositions need to be deferred to run-time which has an impact on performance.

**Volume Measure (VM)**

An important notion in parallel programming is the amount of memory that is needed for the computation. Especially, for programs that have large data structures it is essential that they fit into (local) memory. We therefore introduce the Volume Measure (VM) that returns the total memory requirement for a V-cal program. This measure is defined in Appendix B.

**Computation Measure (CM)**

The computation measure returns the number of operations in a program. This measure is subdivided in a measure for computations in the index domain and for the computations in the data domain. This measure does not give a very precise estimate of the actual computation, but gives an indication of the complexity. It is very useful for the comparison of two versions of the same program, because it can give a good relative measure. The definition of this measure can also be found in Appendix B.

**Optimisation of Renumber Views**

It was already stipulated that composition of special views like Renumber in combination with data-dependent views cannot be resolved at compile-time. In this section, we look at optimisations for these type of compositions. We first consider the general translation of a Renumber view in parameter bindings and selections:

\[
\text{Renumber}(ve) =^vI \Delta(k \leftarrow (dp(\text{Index}(ve)),\text{True})) \{\text{pos(}\text{Index}(ve),k)\} \ve
\]

The CM measure for this expression is very bad, because the size of ve must be determined and the pos function is very expensive, because must retrieve positions of elements. For each element this may take a number of computations that lies in the order of the size of the set. Especially, if the Renumber view is used multiple times in succession, it is advantageous to use an intermediate structure T to store the results of the renumber operation:
\[ \Delta(k \leftarrow (dp(\text{Index}(ve)), True)) \] 
\[ [k]T := [\text{pos}(\text{Index}(ve), k)] \text{ve} \]

The Renumber view is then equal to \( \Delta(k \leftarrow (dp(\text{Index}(ve)), True)) [k]T \) as long as the index set of ve does not change.

**Incremental Optimisation of Renumber**

If the index set of ve does change, we can still optimise the Renumber view in some cases. An example is a view V, that selects all elements of a certain input index set except for a single element X(A) which is computed using shape A:

\[ V = \sqrt{(K(A), \ dp, \ ip)}, \text{ where } K = (b, R(i, j) = (i \in X(A))), \]
\[ dp(l, u) = (l, u), \text{ and } ip(i) = f(i), \text{ and } |X(A)| = 1 \]

The composition of the above view with the Renumber view is not possible, because the latter one is a complex view. We can however, find a reformulation of the view composition that makes use of an index structure T:

**Definition 01**

Consider the views V and W, where \( X \subseteq N^b \):

\[ V = \sqrt{(K(A), \ dp, \ ip)} \]
\[ K = (b, R(i) = (i \in X(A))), \]
\[ dp(l, u) = (l, u), \text{ and } ip(i) = f(i) \text{ and } |X(A)| = 1 \]

\[ W = \sqrt{(K, \ dp, \ ip(T))} \]
\[ K = (b, True), \ dp(l, u) = (l, u-1), \ ip(i) = [f(i)]T \]

Let \([i]T := i\) initially, then

\[ \text{Renumber}(V(A)) \iff \]
\[ ([0]T, [X(A)]T) := ([X(A)]T, [0]T)); \text{W}(A) \]

In the above definition the forms (1) and (2) have the following Index Overhead, Volume, and Computation Measures:

\[ IM(\ (1) ) = \text{size}(N^b)/(\text{size}(N^b)-1) = 1 \]
\[ IM(\ (2) ) = 1 \]
\[ VM(\ (1) ) = \text{size}(T) = \text{size}(b) \]
\[ VM(\ (2) ) = \text{size}(T) = \text{size}(b) \]
\[ CM_f(\ (1) ) = \text{size}(N^b)*CM(\text{pos})+ CM(f) + CM(X(A)) \]
\[ CM_f(\ (2) ) = CM(f) + CM(X(A)) \]

The difference of the two alternatives is \( \text{size}(N^b)*CM(\text{pos}) \) for each iteration, and this is a significant optimisation of the computation complexity. This also illustrates the use of measures for automatic transformation.

The type of optimisations described in definition 01 are very hard to deduce in any formal system and they constitute, what a programmer might refer to as “tricks”. The number of such “tricks” is restricted and is more or less based on a working set of Booster programs that use these type of views. If such an optimisation is not found, a price is paid in terms of “run-time” overhead to
implement the *Renumber* view. This particular optimisation will be illustrated in the Gaussian Elimination example in the next section.

### 7.2.3 Ordering Optimisations

The ultimate goal for a *Booster* program is to execute as many operations in parallel as possible. In the optimisation process of *V-cal* programs we therefore need a measure for the degree of parallelism in the program. This measure is referred to as the *Parallel Measure* (PM) and is defined in Appendix B.

Note that the *Computation Measure* (CM) divided by the *Parallel Measure* (PM) estimates the maximal obtainable speed-up for the program \(P\):

\[
\text{Speedup}(P) \leq \frac{\text{CM}(P)}{\text{PM}(P)}
\]

### 7.2.4 Machine Specific Optimisations

Specific operations, most notably vector operations, may execute much more efficiently on a machine that has vector processing capabilities. Furthermore, if data is stored in memory such that it can be retrieved in one stride, this may also result in much faster programs. We therefore introduce a *Machine Measure* (MM) that takes a view expression as an argument and returns an estimate of the execution time. It should be noted that the computation of this measure may in some cases incur the simulation of a part of the program.

Below, we describe two example of techniques for optimising the computation for a parallel machine with vector processing capabilities. The first optimisation introduces vector optimisations in a program. The last optimises memory access behaviour.

**Vector-optimisations**

The use of vector-instructions on a (parallel) machine that has vector-processing capabilities may significantly increase performance. In *V-cal* these types of optimisations are described by reduction-rules involving superscalar functions. For example, by applying reduction rule \(VI2\) a scalar operation can be transformed to a vector-operation.

\[
\Delta(i \leftarrow I) \#e ([f(i)]A, [g(i)]B) = \#d(\Delta(i \leftarrow I) [f(i)]A, \Delta(i \leftarrow I) [g(i)]B)
\]

**Strip-mining**

Strip-mining is used to split a large vector in a collection of smaller vectors such that each part of the vector fits into the vector-register. The rule is defined as follows, where \(c\) is the size of the vector register with \(c \in (0:n)\) and \(c \mod n = 0\), then

**Definition O2**

\[
\Delta(i \leftarrow (0:n)) ve(i) = \Delta((k,l) \leftarrow (0:x) \times (0:(n \div c)-1)) ve(k+l\times c)
\]
7.3 Example Translations and Optimisations

The translation and optimisation of Booster programs and their annotations is illustrated by two examples. The first is the translation of various versions of the summation of all the indices of a matrix. The second example is the Gaussian elimination without pivoting and the last Gaussian elimination with pivoting.

7.3.1 Space-Time Trade-offs

Consider the following Booster program:

\begin{verbatim}
SHAPE T (1) OF INT;
V { n # n } [i, j] <- i+j;
T := reduce(+, V);
\end{verbatim}

Consider the following V-cal program fragment which allows for the demonstration of a number of different reductions, where \(c = n - 1\)

\[\Delta(i, j) \leftarrow (0:x \times 0:x) \quad \text{// } \{ i,j \} V := i + j; \]
\[\{ T := reduce(+, \Delta(i, j) \leftarrow (0:x \times 0:x)) [i, j] V; \}\]

The first reductions involve the rewriting of the reduce-operation (Proof is not given here) and parameter decomposition (Theorem V6)

\[\Delta(i \leftarrow (0:x)) \quad /\quad \Delta(j \leftarrow (0:x)) \quad \text{// } \{ i,j \} V := i + j; \]
\[\{ T := reduce(+, \Delta(i \leftarrow (0:x)) \quad reduce(+, \Delta(j \leftarrow (0:x))) [i, j] V; \}\]

The next reduction step introduces an explicit intermediate variable \(S\) with index set \(0:x\):

\[\Delta(i \leftarrow (0:x)) \quad /\quad \Delta(j \leftarrow (0:x)) \quad \text{// } \{ i,j \} V := i + j; \]
\[\Delta(i \leftarrow (0:x)) \quad /\quad \{ S[i] := reduce(+, \Delta(j \leftarrow (0:x))) [i, j] V \}
\[T := reduce(+, \Delta(i \leftarrow (0:x))) [i] S\]

The two first clauses involving two parameter bindings can be merged (Definition C5):

\[\Delta(i \leftarrow (0:x)) \quad * \quad \text{// } \{ j V := i + j; \}
\[\{ \Delta(j \leftarrow (0:x)) \quad /\quad S[i] := reduce(+, \Delta(j \leftarrow (0:x))) [j] V; \}
\[T := reduce(+, \Delta(i \leftarrow (0:x))) [i] S;\]

The total memory requirement of the last scheme is lower than that of the first at the expense of the minimal computation time (the parallel measure \(PM\)).
7.3 EXAMPLE TRANSLATIONS AND OPTIMIZATIONS

7.3.2 Gaussian Elimination without Pivoting

The Booster version of the Gaussian elimination algorithm allows us to illustrate a number of aspects of translating to and optimising programs in V-cal. The algorithm involves the translation of a data-dependent view function and of a recursive view identifier definition. Here $A$ is a shape with index set $(0:n \times 0:n)$:

(1) VIEW FUNCTION Pivoting (Q) -> (_P, _R, _C, _B);
(2) Q (m \# m); _P(1); _R, _C (m-1); _B (m-1 \# m-1);
(3) BEGIN
(4) _P <- Q[0, 0];
(5) _R <- Q[0, 1:upb];
(6) _C <- Q[1:upb, 0];
(7) _B <- Q[1:upb, 1:upb];
END;
(8) H <- A;
(9) WHILE SIZE(H) > 1 DO
(10) (P,R,C,B) <- Pivoting(H);
(11) B[i,j] := B[i,j] - (C[i]/P)*R[j];
(12) H <- H[1:upb,1:upb];
END;

The translation of this algorithm is split in two: the translation of the view function Pivoting and the algorithm itself. The view function has four output identifiers which are expressed in terms of the input identifier $Q$. (For convenience, we will use the following denotation for views: $\sqrt{(b, R, dp(l,u) = (0:f(u)), ip(i) = g(i)) = \langle f(l), g(l) \rangle}$)

\[
P = \langle..., (0,0)\rangle H \\
R = \langle i-1, (0,i+1)\rangle H \\
C = \langle j-1, (i+1,0)\rangle H \\
B = \langle (i-1,j-1), (i+1,j+1)\rangle H
\]

Secondly, in the algorithm a recursive view definition is used based on view identifier $H$. This view identifier $H$ is specified recursively as follows, where $V$ is the view that results from translating the view statement in line (12).

$H^1 = A$;

\[H^k = Renumber(V(H^{k-1})), \text{ where } V = \langle (i-1,j-1), (i+1,j+1) \rangle\]

The domain propagation function of $V$: $dp(l,u) = (0:f(u))$ yields a zero-based consecutive index set and therefore the Renumber view can be eliminated.
Furthermore, given the fact that both \( ip \) and \( dp \) are data-independent the definition of \( H \) can be written in terms of \( V^k \): (Theorem 6.20)

\[
H^k = V^k(A), \text{ where } V^k = \langle (i-k,j-k), (i+k,j+k) \rangle
\]

Having this single expression for \( H \) we can express the view identifiers \( P, R, C, \) and \( B \) in these terms, using view composition and application (V9):

\[
P^k = \langle \ldots, (0,0) \rangle \langle (i-k,j-k), (i+k,j+k), A = \langle \ldots, (k,k) \rangle \rangle A
\]

\[
R^k = \langle \ldots, (1,0) \rangle \langle (i-k,j-k), (i+k,j+k), A = \langle \ldots, (j-k-1), (k,i+k+1) \rangle \rangle A
\]

\[
C^k = \langle \ldots, (i-1,0) \rangle \langle (i-k,j-k), (i+k,j+k), A = \langle \ldots, (i-k), (i+k+1,k) \rangle \rangle A
\]

\[
B^k = \langle \ldots, (i-1,j-1), (i+1,j+1) \rangle \langle (i-k,j-k), (i+k,j+k), A = \langle \ldots, (i-k-1,j-k-1), (i+k+1,j+k+1) \rangle \rangle A
\]

Substitution of these view identifiers in the translated content statement in line (11) yields the following \( V\)-cal program, where \( \forall (id, ip(i,j) = (i,j)) \) and \( \forall (id, ip(i,j) = i) \):

\[
\text{While Size}(B^k) > 1 \{
\begin{align*}
\langle \ldots, (i,j) \rangle & \langle (i-k-1,j-k-1), (i+k+1,j+k+1) \rangle \rangle A := \\
\langle \ldots, (i,j) \rangle & \langle (i-k-1,j-k-1), (i+k+1,j+k+1) \rangle \rangle A - \\
\langle \ldots, (i,0) \rangle & \langle (i-k), (i+k+1,k) \rangle \rangle A \langle \ldots, (k,k) \rangle \rangle A * \\
\langle \ldots, (0,j) \rangle & \langle (j-k-1), (k,i+k+1) \rangle \rangle A; 
\end{align*}
\}
\]

The function used in the condition of the While can be simplified: \( \text{Size}(B^k) = \text{Size}(dp^k((0:n \times 0:n))) = \text{Size}((0:n-k-1) \times 0:n-k-1)) = (n-k)^2 \). The condition \( \text{Size}(B^k) > 1 \) holds iff \((n-k)^2 > 1\). This implies that \( n-k = 0 \) or \( n = k \) is the stop criterion. We can therefore replace the While by the parameter binding \( \Delta(k \leftarrow K) * \), where \( K = (0:n) \). Composition of all views and application of Theorem V1 yields:

\[
\Delta(k \leftarrow K) * \{
\begin{align*}
\Delta(i \leftarrow (0:n-k-1)) \Delta(j \leftarrow (0:n-k-1)) [i+k+1,j+k+1]A := \\
\Delta(i \leftarrow (0:n-k-1)) \Delta(j \leftarrow (0:n-k-1)) [i+k+1,j+k+1]A - \\
\Delta(i \leftarrow (0:n-k-1)) \Delta(j \leftarrow (0:n-k-1)) [i+k+1,k]A[k,k]A * \\
\Delta(i \leftarrow (0:n-k-1)) \Delta(j \leftarrow (0:n-k-1)) [k,j+k+1]A; 
\end{align*}
\}
\]

Extraction of the parameter bindings from the operators – and * (Theorem V12) yields:

\[
\Delta(k \leftarrow K) * \{
\begin{align*}
\Delta(i \leftarrow (0:n-k-1)) \Delta(j \leftarrow (0:n-k-1)) [i+k+1,j+k+1]A := \\
\Delta(i \leftarrow (0:n-k-1)) \Delta(j \leftarrow (0:n-k-1)) \\
[i+k+1,j+k+1]A - [i+k+1,k]A[k,k]A * [k,j+k+1]A; 
\end{align*}
\}
\]
7.3 Example Translations and Optimizations

Extraction of the parameter bindings from the assignment (C2) yields:

\[
\Delta(k \leftarrow K) * \{ \\
\Delta(i \leftarrow (0:n-k-1)) // \Delta(j \leftarrow (0:n-k-1)) // \\
\{[i+k+1,j+k+1]A := [i+k+1,j+k+1]A - [i+k+1,k]A/k,k]A * [k,j+k+1]A; \}
\}
\]

We have now obtained a translated and normalised version of the Booster program.

7.3.3 Gaussian Elimination with Pivoting

The example with pivoting in Booster equals:

VIEW FUNCTION Pivoting (Q) -> (\_P, \_R, \_C, \_B); 
Q (m \# m); \_P (1); \_R, \_C (m-1); \_B (m-1 \# m-1); 
BEGIN 
  INDEX pivot = maxindex(abs(Q[\_, 0])); 
  \_P <- Q[pivot, 0]; 
  \_R <- Q[pivot, 1:upb]; 
  \_C <- Q[\pivot, 0]; 
  \_B <- Q[\pivot, 1:upb]; 
END;

The view identifiers \_P, \_R, \_C, and \_B now equal

- \_piv = reduce(maxindex, \_Q(k)) \_Q(k,0)H
- \_piv = <..., (pivot,0)> H
- \_R = <i-1, (pivot,i+1)> H
- \_C = <R(i) = (i \neq pivot ), j-1, (i+1,0)> H
- \_B = <R(i,j) = (i \neq pivot ), (i-1,j-1), (i+1,j+1)> H

Again, we translate the recursively defined view identifier \_H^k:

- \_H^1 = A;
- \_H^k = renumber(V(H^{k-1})), where
  - V = (R(i,j) = (i \neq pivot ), dp(i,j) = ((0:i-1),(0:j-1)), ip(i,j) = (i+1,j+1)), with
  - pivot = reduce(maxindex, \Delta(k \leftarrow Index(H^{k-1}))) abs([k,0]H^{k-1}

A problem in this case, however, is that we cannot resolve \_V^k in \_H^k = \_V^k(A), because \_V is content-dependent. As a consequence the translation of the Booster program yields a V-cal program, where the view is to be computed at every iteration. To optimise this process, we apply Theorem 01, obtaining:

- pivot = reduce( maxindex, \Delta(k \leftarrow Index(H^{k-1}))) abs([k,0]H^{k-1});
- ([0]T, [pivot]T = ([pivot]T, [0]T);
- \_H^k(A) = V(A) = (dp(i,j) = ((0:i-1),(0:j-1)), ip(i,j) = (T[i+k+1],j+k+1))A
Substitution of this view in the program yields, analogous to the previous example (Elimination without pivoting)

\[ \Delta(i \leftarrow (0:n)) \quad / \quad \{ [i]T := i \} \; ; \]
\[ \Delta(k \leftarrow K) \quad * \quad \{ \]
\[ \text{pivot} = \text{reduce}(\text{maxindex}, \Delta(k \leftarrow \text{Index}(H^{k-1})) \; \text{abs}([[k]T,0])H^{k-1}) ; \]
\[ ([k]T, [pivot]T = ([pivot]T, [k]T) ; \]
\[ \Delta(i \leftarrow (0:n-k)) \quad / \quad \Delta(j \leftarrow (0:n-k)) \quad / \quad \}
\[ \{ [i,j] \Delta((x,y) \leftarrow (0:n-k-1 \times 0:n-k-1)) [[x+k+1]T,y+k+1]A := \]
\[ [i,j] \Delta((x,y) \leftarrow (0:n-k-1 \times 0:n-k-1)) [[x+k+1]T,y+k+1]A - \]
\[ [i] \Delta(y \leftarrow (0:n-k-1)) [[x+k+1]T,k]A / [[k]T,k]A * \]
\[ [j] \Delta(x \leftarrow (0:n-k-1)) [[k]T,x+k+1]A ; \} \]}

Contraction of selection and parameter bindings yields the normalised V-cal program for Gaussian elimination with pivoting:

\[ \Delta(i \leftarrow (0:n)) \quad / \quad \{ [i]T := i \} \; ; \]
\[ \Delta(k \leftarrow (0:n)) \quad * \quad \{ \]
\[ \text{pivot} = \text{reduce}(\text{maxindex}, \Delta(k \leftarrow (0:n-k-1)) \; \text{abs}([[k]T,0])A) ; \]
\[ ([k]T, [pivot]T = ([pivot]T, [k]T) ; \]
\[ \Delta(i \leftarrow (k:n-1)) \quad / \quad \Delta(j \leftarrow (k:n-1)) \quad / \quad \}

From this last program it is clear that this program can be translated to for example a C or Fortran program. The parameter bindings with sequential orderings translate to sequential loops and parameter bindings with parallel orderings translate to parallel loops. The reduce construct translates to a loop and a condition-statement in which the maximal index is computed.
Chapter 8
Generation of Parallel Programs

We now come to discuss the semantics and translation of Booster programs in conjunction with its annotations and the generation of parallel programs. On this level, a Booster program and its computation and data organisation are integrated and a corresponding parallel version including data transport is generated. Here, the duality of (implicit) parallel programs with annotations and parallel programs with explicit parallelism becomes apparent.

As was discussed in Chapter 3 the parallel programs are generated using the so-called SPMD (Single Program Multiple Data)-principle. According to this principle, every processor loads the same — single — program but executes a different set of instructions on a distinct set of data, depending on its processor number.

In this chapter, we will discuss the transformation of V-cal programs to SPMD-type parallel programs on a clause by clause basis. Translation aspects of groups of clauses will only be touched upon here, since these are merely an extension of the clause-by-clause principle. Furthermore, we have separated transformations of V-cal clauses dealing with annotations on shapes from those dealing with annotations on views and view identifiers. The transformations and the introduction of data transport for the latter category is more complicated than for shapes.

One step in the translation process, namely the translation of annotation specifications in the annotation language to V-cal clauses, has been skipped. The syntax of these annotations closely resembles the syntax of view statements in the Booster language itself and the same translation approach can be taken. The discussion of the translation of Booster programs to parallel programs is organised as depicted in Figure 24.

In Section 8.1, the machine-dependent part of V-cal is introduced which is used to express data transport and representation of data in memories. In Section 8.2, we discuss the integration of Booster programs and annotations to determine computation responsibility. In Section 8.3, we elaborate on the translation of programs with respect to the management of data. Section 8.4 describes a number of optimisations that can be derived for index sets. Optimisations for the transport of data is discussed in Section 8.5. In Section 8.6, performance prediction and automatic parallelisation of Booster are dis-
discussed shortly. We conclude the chapter with a number of examples in Section 8.7.

![Diagram of Booster programs and annotations]

**Program Template**

For the derivation of certain results in the coming sections and chapters, we will need a general denotation for an arbitrary clause, where we are only interested in the *Defs* and *Refs*, which were defined in Chapter 6. The denotation is:

{Clause \{ Def-list | Ref-list \}}

For example:

\[
\text{if } A > 0 \text{ then } \Delta(i \leftarrow I) \diamond \{ [i]B := [2^*i]C \} \\
\quad \Rightarrow \Delta(i \leftarrow I) \diamond \text{Clause} \{ [i]B \mid A, [2^*i]C \}
\]

\[
\Delta(i \leftarrow I) \diamond \{ [i]A := [2^*i]B + [2^*i]C \} \\
\quad \Rightarrow \Delta(i \leftarrow I) \diamond \text{Clause} \{ [i]A \mid [2^*i]B, [2^*i]C \}
\]

In case reductions are used in the clause, the scheme may involve parameter bindings in the reference section:

\[
\Delta(i \leftarrow I) \diamond \{ [i]A := \text{reduce}(+, \Delta(k \leftarrow K)) [2^*k,i]C \} \\
\quad \Rightarrow \Delta(i \leftarrow I) \diamond \text{Clause} \{ [i]A \mid \Delta(k \leftarrow K) [2^*k,i]C \}
\]
8.1 Introducing Parallelism in V-cal

The part of V-cal introduced in the previous chapter is concerned with expression of algorithms in terms of shapes. The integration of algorithms and annotations, however, requires shapes to be expressed in terms of specified annotations. The expression of annotations in V-cal is described in the first section.

Once the annotations are integrated in the V-cal program it is necessary to introduce data transport in the form of Get and Put operations. Depending on the addressing facilities provided by the programming model these Get and Put operations can be specialised to either a Load/Store-model (global address space) or a Send/Receive-model (distributed address space). These models are introduced in Section 8.1.3.

Central in optimising parallel programs is the decomposition of the program and the minimization of data transport overhead. In order to define these optimisations we need information about which processor is responsible for which memory and what the cost of transporting data from memory to processor is.

8.1.1 Interconnection Networks

The notion of an interconnection network is represented in V-cal by two notions: access responsibility and cost:

- The responsibility relation \( \text{resp}(l) \) defines which processor(s) are able to access a certain location \( l \);
- The cost function \( \text{cost}(p,l) \) defines the relative cost for obtaining data from a location \( l \) accessible by a certain processor \( p \).

For example, a shared-memory parallel machine has equal distance to memories for all processors and all processors are able to access the memory with equal cost. A distributed-memory machine with local memory and a linear interconnection network has a distance function \( \text{cost}(p,l) = (\text{resp}(l) - p) \mod p_{\text{max}} \) and a function \( \text{resp} = \text{id} \), whereas the cost function may vary for local and global memory.

Note that this representation of an interconnection network allows for a wide range of architectures. This ranges from a shared-memory to a "paired" distributed-memory architecture to hybrid systems with combinations of local and shared memory. We will return to this in the next section.

8.1.2 Annotations in V-cal

Annotations in Booster describe the relation between shapes, views, and the underlying architecture. These relations are described with the same concept
as in *Booster*: the view. There is, however, one distinction: annotations in *Booster* can describe data or computation duplication. This implies that each value of an element of shape $A$ may be "stored" on multiple places in a memory structure.

In this thesis, we will not elaborate on data and computation incarnation mappings. We restrict ourselves to (overlapping) computation and data partitioning annotations on views and shapes. However, the same framework as for the partitioning annotations can be used for the integration and translation of these data incarnation annotations.

**Data Organisation Annotations**

Each shape or view may be annotated and this is expressed in terms of a so-called *memory-image*. This is a symbolic reference to the physical memory, but each *memory-image* has a memory-number and position within that memory assigned to it. A *memory-image* of a shape or view $A$ is denoted as $A^M$. The general relation between a data structure and a memory-image is expressed as follows:

$$A = \Delta(q \leftarrow \text{Index}(A)) \text{ChoiceLA}(\Delta(dd \leftarrow DD(q)) \{\text{locationA}(q, dd)\} A^M)$$

The `locationA` function returns a tuple of *memory-number* and *position-number* of the element within that memory: `locationA(i, dd) = (memA(i, dd), positionA(i, dd))`. Parameter `dd` or *data duplication* is argument to the `locationA` function and is bound to the set `DD(i)` defining the number of times the value for $i$ is replicated. Although the same value may be stored multiple times in a memory-structure, one only needs to retrieve a single value. We therefore introduce a `ChoiceL(location)A` function, which is defined as follows in terms of a binary choice function:

**Definition 8.1**

Let $Site$ be the tuple $\text{Var} \times N \times (N \times N)$, then:

$$\text{Choice} : Site \times Site \rightarrow Site$$

$$\text{ChoiceLA}(\Delta(d \leftarrow D) ve(d)) = \text{reduce(Choice, } \Delta(d \leftarrow D)) ve(d)$$

The tuple $Site$ denotes the *variable name*, the *processor-number*, and the *location* = $(\text{mem, position})$. The `Choice` function selects a single site out of two sites, based upon some criterion, such as the cost for transporting a value from a processor that needs a certain value and the location.

**Example 8.1**

An example of an annotation in the annotation language of *Booster* is:
VIRTUAL MACHINE AM ( PROC processor (1), MEM memory (n));
IMPORT Example::A (n);
    A[i] <- [_, i-1:i+1]AM;

The corresponding translation in V-cal is:

\[ A = \Delta(q \leftarrow (0:n)) \text{ChoiceLA}(\Delta(dd \leftarrow DD(q)) [\text{locationA}(q,dd)]A^M) \]

where the following functions and sets are used:

\[ DD(q) = \{ dd \in q-1:q+1 \} \]
\[ \text{locationV}(q,dd) = (dd, q) \]

An example of the Choice function associated with the ChoiceLA function is the following:

\[ \text{Choice}((p,l_1), (p,l_2)) = \text{min}(\text{cost}(p, l_1), \text{cost}(p, l_2)) \]

**Computation Responsibility**

In the context of computation responsibility, shapes and views are assigned to processors. Unlike mappings on memories, mappings on processors do not result in actual storage of data, but determine where to compute data. We distinguish the parameter bindings used for the expression of processor responsibility from others by using the following notation: \( \delta(p \leftarrow P) \), where \( P \) is the set of processors.

Like the data organisation annotation the relation between a data structure and a processor responsibility is expressed as an equivalence and always includes a memory-mapping. The resulting processor-image of a shape or view \( A \) is denoted as \( A^P \):

\[ A = \Delta(i \leftarrow I) \delta(pd \leftarrow PD(i)) \text{ChoiceLA}(\Delta(dd \leftarrow DD(i)) [\text{proc-numberA}(i,pd), \text{locationA}(i,dd)]A^P) \]

The annotation language allows for the definition of duplication of computation responsibility and therefore the parameter \( pd \) or processor duplication is defined, where \( PD(i) \) describes the processor responsibility duplication for the element \( i \).

**Relation between Processor- and Memory-image**

In the sequel of this chapter we need a relation between the processor-image and memory-image of a data structure. For the assignment, we define this relation as follows:
Definition P1
\[ \Delta(k \leftarrow K) \diamond \delta(p \leftarrow P) \equiv \{ A^M := ve(k) \} \iff \Delta(k \leftarrow K) \diamond \{ \delta(p \leftarrow P) [p, f(k)]A^P := ve(k) \} \]

This relation implies that a single assignment that includes a “parallel” parameter binding with the processor-image \( A^P \) is equal to a set of parallel assignments with the memory-image \( A^M \).

8.1.3 Data Transports

The management of data within the (parallel) machine is described by so-called transports. These transports can define memory-to-memory, main memory to cache, or disk-to-memory data transports. In general, we have Get and Put operations, which handle the retrieval and storage of data items. The nature of the data transport may differ, depending on the kind of protocol that is required on a certain virtual machine.

In those cases the data transport operations are refined into two types: the load/store and the send/receive type of transport. Load/store assumes that a processor issuing the calls has access to each desired memory location without any need for intervention by a different processor. Send/receive assumes at least two different processors for the transport of data. The processor responsible for the memory containing the data element needs to issue a Send, whereas the processor that needs the data element issues a corresponding Recv.

Definition 8.2
In addition we have the following clauses in \( V-cal: Site = (V, proc, (mem, position)) \)

\[ \text{Load}(Site) \mid \text{Store}(Site) \mid \text{Send}(Site) \mid \text{Recv}(Site) \]

Relation between operations

The relation between Get and Put on the one hand and Load, Store, Send, and Recv on the other is given below, where \( p \) is the processor-number responsible for performing the operation and \( q \) the processor responsible for the memory that contains the value. We assume the following forms for both Put and Get operations:

\[ \delta(p \leftarrow P) \equiv \delta(q \leftarrow P) \equiv \text{Get}(\Delta(i \leftarrow I) ve(i, p, q)) \]
\[ \delta(p \leftarrow P) \equiv \delta(q \leftarrow P) \equiv \text{Put}(\Delta(i \leftarrow I) ve(i, p, q)) \]

The Load operation only involves a single processor that fetches a set of data-items from a memory to which it has access \((p = q = mynode)\). Here mynode
denotes the processor number of the current processor. If the processor that needs a data-element does not have access to that memory we need a Send operation from the processor $q$ that has access to the memory ($p \neq q = \text{mynode}$) and aRecv operation from the processor $p$ that needs the data ($q \neq p = \text{mynode}$). Consequently, the Get operation expands to$^1$:

\[
\delta(\text{mynode} \leftarrow P) // ( \\
\text{Load}(\Delta(i \leftarrow I) ve(i, \text{mynode}, \text{mynode})) || \\
\{ \delta(p \leftarrow P \setminus \{\text{mynode}\}) // \text{Send}(\Delta(i \leftarrow I) ve(i, p, \text{mynode})) ; \\
\delta(q \leftarrow P \setminus \{\text{mynode}\}) // \text{Recv}(\Delta(i \leftarrow I) ve(i, \text{mynode}, q)) \}
\]

A similar derivation holds for the Put operation, where a Store operation involves a single processor ($p = q = \text{mynode}$). If the processor that needs to store a data-element but does not have access to that memory we need a Send operation from the processor $q$ that has access to the memory ($p \neq q = \text{mynode}$) and aRecv operation from the processor $p$ that stores the data ($q \neq p = \text{mynode}$), yielding:

\[
\delta(\text{mynode} \leftarrow P) // ( \\
\text{Load}(\Delta(i \leftarrow I) ve(i, \text{mynode}, \text{mynode})) || \\
\{ \delta(p \leftarrow P \setminus \{\text{mynode}\}) // \text{Send}(\Delta(i \leftarrow I) ve(i, p, \text{mynode})) ; \\
\delta(q \leftarrow P \setminus \{\text{mynode}\}) // \text{Recv}(\Delta(i \leftarrow I) ve(i, \text{mynode}, q)) \}
\]

For some architectures a simpler scheme exists. In a shared-memory architecture each processor has access to all memories, which implies that $p = q$ for every $i$ and the above scheme simplifies to:

\[
\delta(p \leftarrow P) // \delta(q \leftarrow P) // \text{Get}(\Delta(i \leftarrow I) ve(i)) = \\
\delta(\text{mynode} \leftarrow P) // \text{Load}(\Delta(i \leftarrow I) ve(i)) \\
\delta(p \leftarrow P) // \delta(q \leftarrow P) // \text{Put}(\Delta(i \leftarrow I) ve(i)) = \\
\delta(\text{mynode} \leftarrow P) // \text{Store}(\Delta(i \leftarrow I) ve(i))
\]

8.1.4 Synchronisation

Apart from data transport operations, we also need to synchronise computations to avoid invalid computation results. To this purpose, we introduce a parallel ordering $\mathcal{O}_p$, that synchronises operations across processors:

---

$^1$ Note that the Load operation may be performed in parallel with the Send andRecv operations, but all sends must precede all receives, because we assume non-blocking sends and blocking receives. Any other scheme could result in deadlock.
Definition P2
A parallel ordering is defined as:

\[ \Delta(k \leftarrow K) \odot \delta(p \leftarrow P) // \text{Clause}([f(k)]A | [g(k)]B) \Rightarrow \delta(p \leftarrow P) // \Delta(k \leftarrow K) \odot_p \text{Clause}([f(k)]A | [g(k)]B) \]

When \( \odot \) defines the sequential ordering, the interchange with \( \delta(p \leftarrow P) // \) will yield a parallel ordering to ensure the preservation of the sequential order. When \( \odot \) defines a parallel ordering the interchange will not incur any penalty in terms of synchronisation overhead. We will not elaborate on this subject in this thesis, because the actual implementation of synchronisation is machine dependent. Moreover, the issue of the synchronisation optimisation has less impact on program performance than data transport.

8.2 Computation Responsibility in V-cal

Annotations, as discussed in Chapter 5, are always related to shapes and views of a Booster program and the integration of the program and annotations, comprises of the substitution of these Booster identifiers by their counterparts in the annotation language. After substitution, the rewrite-rules of the calculus from Chapter 6, in combination with new rules defined in this chapter, are used to obtain SPMD versions of the programs defined.

8.2.1 Computation Responsibility for Shapes

We will start with a relatively simple derivation of computation responsibility for shapes to illustrate the reduction-process followed by the general scheme including annotations on views in the next section. The derivation process involves the substitution of the annotation in a program. Next a parallel SPMD program is derived by extracting the parameter binding \( \delta(p \leftarrow P) \).

Annotations

In the case of computation responsibility, we only need to consider the processor allocation part of the annotation and for reasons of clarity we will omit the location specification. Section 8.3 uses this specification to derive data transport schemes. The general annotation template for annotations on shapes with computation duplication equals:

\[ A = \Delta(i \leftarrow I) \delta(pd \leftarrow PD(i)) [\text{proc}(i, pd), \ldots]A^P \]  

(1)

Generalised program model

We consider the following generalised template of a V-cal clause:

\[ \Delta(i \leftarrow I) \odot \text{Clause}([f(i)]A | [g(i)]B) \]  

(2)
which involves a (viewed) shape \([f(i)]A\) that is changed in the clause and (viewed) shape \([g(i)]B\) that is used in the clause and an ordering \(\diamond\) on the elements of the index set \(I\). For the derivation of computation responsibility, the Ref part of the clause is not involved and we therefore omit it in the sequel.

Integration

The structures \(B\) in (2) use annotations according to the same template as \(A\) with functions: \(procB\) and \(locationB\). Integration of annotations and program involves the substitution of the view or shape identifier in (2) by its annotation(1):

\[
\Delta(i \leftarrow I) \Diamond
\]

\[
}\text{Clause}( [f(i)] \Delta(q \leftarrow Q) \delta(pd \leftarrow PD(q)) [\text{procA}(q, pd), \ldots]A^P | \ldots )
\]

(3)

The selection and parameter binding in \(q\) are eliminated by applying Contraction yielding:

\[
\Delta(i \leftarrow I) \Diamond \text{Clause}( \delta(pd \leftarrow PD(f(i))) [\text{procA}(f(i), pd), \ldots]A^P | \ldots )
\]

(4)

SPMD-type transformations

In \(V-cal\) program (4), we have obtained a reduced integrated version of program and annotation. However, the goal of the transformation is to obtain a SPMD-type of program: a single program for each processor that only differs in a parameter \(p\) denoting the processor-number. Hence, our goal is to obtain a \(V-cal\) program of the form:

\[
\delta(p \leftarrow P) /\!\!\!\!/ \text{Clause}(p)
\]

In the derivation process, we use the convention defined in Chapter 5, that every processor \(p\) is responsible for the update of \([p, locationA(f(i))]A^P\) and corresponding computations. From this, we can deduce the following view-expression for the changed term in (4), by observing that \(\text{procA}(f(i), pd) \in P\). Furthermore, by definition the set \((P, \text{procA}(f(i), pd) = p)\) contains one element for every \((i, pd)\). This implies that the size of the latter set is equal to 1 and therefore extraction (Theorem V9) can be applied to (4), yielding:

\[
\Delta(i \leftarrow I) \Diamond \text{Clause}( \delta(pd \leftarrow PD(f(i)))
\]

\[
\delta(p \leftarrow (P, \text{procA}(f(i), pd) = p)) [p, \ldots]A^P | \ldots )
\]

(5)

The SPMD-concept now dictates the variable \(p\) to be filtered to the top most level of the clause, yielding a — single — program that only differs in the parameter \(p\) and which acts upon those data elements that have indices satisfying the
condition \( procA(f(i), pd) = p \). According to theorem P1 \( \delta(pd \leftarrow ...) \) and \( \delta(p \leftarrow ...) \) may be extracted from the Clause:

\[
\Delta(i \leftarrow I) \land \delta(pd \leftarrow PD(f(i))) \land \delta(p \leftarrow (P, procA(f(i), pd) = p)) \land \text{Clause( [..]A^M | ... )}
\]  

(6)

We can now interchange parameter binding \( \delta(p \leftarrow ...) \) with \( \delta(pd \leftarrow PD(f(i))) \) according to C5:

\[
\Delta(i \leftarrow I) \land \delta(p \leftarrow P) \land \delta(pd \leftarrow PD(f(i)), procA(f(i), pd) = p)) \land \text{Clause( [..]A^M | ... )}
\]  

(7)

According to Definition P2, we may interchange parameter bindings \( \Delta(i \leftarrow I) \land \delta(p \leftarrow P) \land \delta(p \leftarrow P) \) and \( \delta(p \leftarrow P) \). For convenience, we define the following set \( \text{Modify}(p) = \{ i \in I \land pd \in PD(f(i)) \mid procA(f(i), pd) = p \} \)

\[
\delta(p \leftarrow P) \land \Delta(i \leftarrow \text{Modify}(p)) \land p \land \text{Clause( [..]A^M | ... )}
\]  

(8)

In (8), we have now obtained a V-cal program that represents the general integrated form of a program and its annotation. A last step is to replace the memory-images of \( A^M \) and \( B^M \) by \( A \) and \( B \) itself, respectively:

\[
\delta(p \leftarrow P) \land \Delta(i \leftarrow \text{Modify}(p)) \land p \land \text{Clause( [f(i)]A | [g(i)]B )}
\]  

(9)

We have now found an expression for the SPMD parallel program for node \( p \) of the parallel machine. Each node \( p \) changes the \( i \) in set \( \text{Modify}(p) \) under condition of the ordering \( \land p \) on \( \text{Modify}(p) \).

**Example 8.2**

An example of such a derivation is the following:

\[
\Delta(i \leftarrow I) \land \{ [2^*i]A := [2^*i]B \ast [3^*i]C \} \land A = \Delta(j \leftarrow (0:n)) \lbrack j \div c, j \mod c \rbrack A^P
\]

Integration yields:

\[
\Delta(i \leftarrow I) \land \{ [2^*i] \Delta(j \leftarrow (0:n)) \lbrack j \div c, ... \rbrack A^P := [2^*i]B \ast [3^*i]C \} \Rightarrow V10a
\]

\[
\Delta(i \leftarrow I) \land \{ [(2^*i) \div c, ... \} A^P := [2^*i]B \ast [3^*i]C \} \Rightarrow V9
\]

\[
\Delta(i \leftarrow I) \land \{ \delta(p \leftarrow (P \lbrack (2^*i) \div c = p \rbrack [p, ... \} A^P := [2^*i]B \ast [3^*i]C \} \Rightarrow P1
\]

\[
\Delta(i \leftarrow I) \land \delta(p \leftarrow (P \lbrack (2^*i) \div c = p \rbrack \} \} \lbrack [2^*i \mod c]A^M := [2^*i]B \ast [3^*i]C \} \Rightarrow P2
\]

\[
\delta(p \leftarrow P) \land \Delta(i \leftarrow I \land (2^*i) \div c = p)) \ast \{ [2^*i \mod c]A^M := [2^*i]B \ast [3^*i]C \}
\]

A last optimisation is the simplification of the index sets of the parameter binding \( \Delta(i \leftarrow I \land (2^*i) \div c = p)) \), which could also be written as \( \Delta(i \leftarrow (c*p) \div 2;((c-1)*p) \div 2-1)) \) for certain values of \( c \). We will return to the optimisation of index sets in Section 8.4.
Another example annotation uses the same program, both with an annotation with computation duplication:

$$A = \Delta(j \leftarrow (0:n)) \delta(pd \leftarrow P) [pd, j]A^P$$

In this case each element of A is to be computed on every processor. A comparable derivation process yields the following form:

$$\Delta(i \leftarrow I) \parallel ( [2*i] \Delta(j \leftarrow (0:n)) \delta(pd \leftarrow P) [pd, j]A^P : = [2*i]B \ast [3*i]C ) \Rightarrow V10a$$

$$\Delta(i \leftarrow I) \parallel ( \delta(pd \leftarrow P) [pd,2*i]A^P : = [2*i]B \ast [3*i]C ) \Rightarrow P1$$

$$\Delta(i \leftarrow I) \parallel \delta(pd \leftarrow P) \parallel ( [2*i]A : = [2*i]B \ast [3*i]C ) \Rightarrow P2$$

$$\delta(pd \leftarrow P) \parallel \Delta(i \leftarrow I) \bullet ( [2*i]A : = [2*i]B \ast [3*i]C )$$

The resulting expression describes the effect of the annotation: each processor executes the complete computation.

Summary

The existing reduction-rules for V-cal as described in Chapter 6 also apply for the derivation of computation responsibility in V-cal. As shown in the above example this process can be automated and applied to any type of V-cal program with annotations on shapes. The example has also shown that the key to efficient solutions to the computation responsibility determination lies in the optimisation of the resulting parameter bindings' index sets. In the automated process the Index Overhead Measure (IM) quantifies the effect of optimisations.

Furthermore, the integration of annotations and the subsequent derivation of the parallel program also changes the Parallel Measure (PM), because for example the interchange of $\delta(p \leftarrow P) \parallel$ and $\Delta(i \leftarrow I) \parallel$ in the last example changes the ordering for $I$ from parallel to sequential, because each SPMD-program on a certain node is executed sequentially. If the size of $I$ is larger than the size of $P$, PM will become larger, indicating that potential parallelism has been lost.

8.2.2 Computation Responsibility for Views

With the computation responsibility annotations for views we derive general computation responsibility by using a template consisting of (dynamic) view identifiers and annotations with computation duplication. This derivation is more complicated than the previous one because of the dynamic nature of views and the associated change in computation responsibility. In addition, annotations with computation duplication allow multiple processors to compute the same result.
General Program Model
Consider the following general program model with annotations on (dynamic) views.

\[ V^0 = \sqrt{\text{initA}}(b_2, R_2, dp_2, ip_2) A \]
\[ \Delta(k \leftarrow K) \cdot \{ \quad V^k = (\sqrt{\text{repeatA}}(b_1, R_1, dp_1, ip_1))V^{k-1}; \]
\[ \quad \Delta(i \leftarrow I(k)) \circ \text{Clause}([f(i)]V^k \mid \ldots) \} \] (1)

For some parameter \( k \) that increases in every iteration, the view on \( A \) changes. \( V^k \) consists of some initial view \( \sqrt{\text{initA}} \) on shape \( A \), followed by a view \( \sqrt{\text{repeatA}} \) that is composed with the previous \( V^{k-1} \). This view identifier is then used in a clause, and consequently changes every iteration.

View Composition
In the general scheme above there still remains an unresolved view composition \( V^k \). In the ideal case, compile-time resolution of view compositions is possible, but this is not always the case. In the special case, where predicate \( R_1 \) is equal to \( \text{True} \) and \( dp_1(b_1) \subseteq b_1 \), we can apply Theorem 6.4 and Theorem V4, to obtain an expression for \( V^k \) in terms of \( A \):

\[ V^k = \sqrt{\text{repeatA}}(b_1, R_1, dp_1, ip_1)\circ \sqrt{\text{initA}}(b_2, R_2, dp_2, ip_2) A = \]
\[ \sqrt{\text{repeatA}}(dp^{k-1}(b_1), \text{True}, dp_1^k, ip_1^k) \circ \sqrt{\text{initA}}(b_2, R_2, dp_2, ip_2) A = \]
\[ \sqrt{\text{V}}(dp^{k-1}(b_1) \land dp_1^k(b_2), R_2 \cup ip_1^k, dp_1^k \circ dp_2, ip_2 \circ ip_1^k) A = \]
\[ \sqrt{\text{V}}(b_\text{V}(k), R_\text{V}(k), dp_\text{V}(k), ip_\text{V}(k)) A \]

Substitution of this resolved view composition yields:

\[ \Delta(k \leftarrow K) \circ \Delta(i \leftarrow I(k)) \circ \{ \]
\[ \quad V^k = \sqrt{\text{V}}(b_\text{V}(k), R_\text{V}(k), dp_\text{V}(k), ip_\text{V}(k)) A; \text{Clause}([f(i)]V^k \mid \ldots) \} \] (2)

Restriction
In practice not every view composition can be rewritten at "compile-time" in the above form. An example of this is the content-dependent view. For the sake of clarity, however, we will use the composed version. This restriction does not violate the generality of the derivation, because if the views cannot be eliminated at "compile-time" they are to be "computed" at "run-time". We will therefore use (2) in the general derivation scheme below.

Integration
On this general dynamic scheme, the following annotation on view \( V^k \), expressed in terms of its machine image \( V^P \), is defined, including the ChoicePV function that selects one element of the duplication per processor.

\[ V^k = \Delta(q \leftarrow I) \delta(pd \leftarrow PD(q)) [\text{procV}(k, q, pd), \ldots]V^P \] (3)
This annotation (3) is integrated in the general clause (2):

$$
\Delta(k \leftarrow K) \bullet \Delta(i \leftarrow l(k)) \Diamond \{ \forall k = \sqrt{\lambda \forall b_V(k)R_V(k)dp_V(k)ip_V(k)A;}
\text{Clause}(f[i]) \Delta(q \leftarrow I) \delta(pd \leftarrow PD(q)) [\text{proc} V(k,q,pd), ...] V^p | ... ) \}
$$

From this point on the same steps are taken as for the derivation of annotations on shapes (steps (3) to (8) in the shape derivation scheme):

$$
\Delta(k \leftarrow K) \bullet \delta(p \leftarrow P) /\delta(pd \leftarrow PD(f(i)) \land \text{proc} V(k,f(i),pd) = p) /\Delta(i \leftarrow I(k)) \circ_p \text{Clause}([...V^k | ... ])
$$

We define the set, which is equal to the previously defined $\text{Modify}(p)$ set, but is extended with a parameter $k$.

$$
\text{Modify}(k, p) = \{ i \in I \land pd \in PD(f(i)) \mid \text{proc} A(k,f(i),pd) = p \}
$$

The last step is to filter the expression $\delta(p \leftarrow P) /$ to the right:

$$
\delta(p \leftarrow P) /\Delta(k \leftarrow K) \bullet_p \Delta(i \leftarrow \text{Modify}(k, p)) \circ_p \text{Clause}([ip_V(k,f(i))]) A \mid ... )
$$

Note that in (6) two orderings — $\Delta(k \leftarrow K)$ and $\Delta(i \leftarrow \text{Modify}(k, p))$ — restrict the parallelism and must synchronise.

This concludes the section on deriving computation responsibility by integrating $V$-cal programs and annotations. In the next section, we will elaborate on the consequences of annotation integration for the management of data.

**Example 8.3**

In this example, taken from Section 5.4.2, we illustrate the derivation of (dynamic) computation responsibility for views.

$$
\begin{align*}
V & = A; \\
\Delta(k \leftarrow K) \bullet \{ \\
\Delta(i \leftarrow \text{Index}(V)) /\{ [i]V^k := [i]V^{k*2}; \\
V^k = \sqrt{(dp(i) = i-1, ip(i) = i+1)V^{k-1}} \\
\} \} & \text{V} = \Delta(j \leftarrow (0:n-k)) [j \div c_b, j \mod c_k]V^M \\
\text{where } c = (n-k) \div p_{\max}
\end{align*}
$$

It follows that $V^k = \sqrt{(dp(i) = i-1, ip(i) = i+1) A = \sqrt{(dp(i) = i-k, ip(i) = i+k) A}$. Consequently $\text{Index}(V^k) = (0:n-k)$. (Below, we eliminate the view statements and we will skip a number of derivation steps, because they were already demonstrated in the previous sections):

$$
\begin{align*}
\Delta(k \leftarrow K) \bullet \Delta(i \leftarrow (0:n-k)) / \\
\{ [i] \Delta(j \leftarrow (0:n-k)) [j \div c_b, ...]V^M := [i]V^{*2}; \} & \Rightarrow V^{10a}
\end{align*}
$$

$$
\begin{align*}
\Delta(k \leftarrow K) \bullet \Delta(i \leftarrow (0:n-k)) / [i \div c_k ...]V^M := [i]V^{*2}; & \Rightarrow V^9 \\
\Delta(k \leftarrow K) \bullet \Delta(i \leftarrow (0:n-k)) / \delta(p \leftarrow (P \mid i \div c_k = p)) / [i]V := [i]V^{*2}; & \Rightarrow C^5
\end{align*}
$$
\[ \Delta(k \leftarrow K) \cdot \delta(p \leftarrow P) /\Delta(i \leftarrow (0:n-k \mid i \text{ div } c_k = p)) \cdot \{ [i]V := [i]V*2; \} \Rightarrow P2 \]
\[ \delta(p \leftarrow P) /\Delta(k \leftarrow K) \cdot p \]
\[ \Delta(i \leftarrow (0:n-k \mid i \text{ div } c_k = p)) \cdot \{ [i]V := [i]V*2; \} \Rightarrow \text{ View Subst.} \]
\[ \delta(p \leftarrow P) /\Delta(k \leftarrow K) \cdot p \Delta(i \leftarrow (0:n-k \mid i \text{ div } c_k = p)) \cdot \{ [i+k]A := [i+k]A*2; \} \]

Summary

The derivation of computation responsibility for annotations on views resembles the derivation on shapes. The main difference is the function \( procV \) that also depends on a parameter \( k \) that may change during execution of the program. This may cause the assignment of computation responsibility to change. This was already discussed in Section 5.5.1, where the programmer can use these type of annotations to achieve load-balancing in the algorithm. The Parallel Measure \( (PM) \) quantifies this effect for the compiler as well as the programmer.

8.3 Data Management

The specification of computation and data organisation on Booster programs is orthogonally defined in the annotation language. In the integration and translation process, however, they do interfere. If the computation of data element \( A \) is assigned to a certain processor, it may need elements of \( B \) for that computation. A memory not directly accessible to this processor may store these elements of \( B \) and therefore this computation organisation may induce transfer of data. On the other hand, explicit data organisations also induce data transport. We will discuss these two types of data transport.

In this section, we distinguish between the derivation of data transport from data organisation annotations on shapes and views. Although we could use the results of annotations on views for shapes by assuming an identity view, we will use the annotation on the shape to illustrate overlapping annotations and expansion of Put and Get operations to obtain Load, Store, Send, and Receive operations.

8.3.1 Data Transport for Shapes

Consider the following general program template on the left\(^2\) that has no notion of explicit parallelism. From the left template, we have to deduce a SPMD-template as given on the right:

\(^2\) Note that the ordering operator for this program must be parallel (\(/\)). If there is a intra-clause dependence the validity of data may be destroyed when a Put or Get operation is performed in between. Moreover, a clause may use multiple variables, but this single shape \( B \) is merely used to illustrate the principle.
Get( Δ(ι ← I) [g(ι)]B );
Δ(ι ← I) // Clause( [f(ι)]A | [g(ι)]B);
Put( Δ(ι ← I) [f(ι)]A )

δ(p ← P) // { Getp( ... ) ;
Δ(ι ← I(p)) // Clause(...);
Putp( ... ) }

The SPMD paradigm dictates that each processor must execute the Get and Put for the elements it uses or changes. We therefore need to use the derivation of computation responsibility that was given in the previous sections and we need the data organisation annotation on B:

\[ B = δ(q ← Index(B)) ChoiceLB( Δ(dd ← DD(q)) [locationB(q,dd)]B^M) \]

The above annotation suffices for the Get operation, because elements of B are retrieved from its memory image B^M. In the Put operation, however, stores elements of A in A^M. For example, the value of a single data item in A may be stored in multiple places in A^M and we therefore need A^M described in terms of A:

\[ Δ(i ← Index(A)) Δ(dd ← DD(i)) ([locationA(i,dd)]A^M = [i]A) \]

This "annotation" is a slight modification of the original annotation and we can still use the same functions and sets.

**Derivation of Get**

The elements Ref( [g(ι)]B ) must be retrieved by the processor responsible for the computation of [f(ι)]A. As was shown in Section 8.2.1, each processor p is responsible for computing elements ModifyA(p) of I. Consequently, processor p must retrieve the following elements of B:

\[ Δ(i ← ModifyA(p)) [g(ι)]B \]

We substitute the annotation for B and we obtain the following Getp operation:

\[ δ(p ← P) // Getp( δ(i ← ModifyA(p))
[g(ι)] Δ(q ← Index(B)) ChoiceLBp( Δ(dd ← DD(q)) [locationB(q,dd)]B^M) \]

After contraction we obtain the following form:

\[ δ(p ← P) // Getp( δ(i ← ModifyA(p))
ChoiceLBp( Δ(dd ← DD(g(ι))) [locationB(g(ι),dd)]B^M ) \]

This last form describes which elements of B are to be retrieved from where and by which processor.

To derive the Load, Send, and Receives from this Get operation, we have to extract an expression that yields the processors that are responsible for accessing the memory on which a certain element of B is located. We therefore extract
the parameter binding \( \delta(q \leftarrow (P, \text{resp}(\text{locationB}(g(i),dd)) = q)) \) that returns all processors \( q \) responsible for \( \text{locationB}(g(i),dd) \):

\[
\text{Get}_{p, q}(\Delta(i \leftarrow \text{Modify}(p)) \text{ChoiceLB}_{p, q}(\Delta(dd \leftarrow DD(g(i))))
\]

\[
\delta(q \leftarrow (P, \text{resp}(\text{locationB}(g(i),dd)) = q)) [\ldots]^{B^M} = V^{16}
\]

\[
\text{Get}_{p, q}(\Delta(i \leftarrow \text{Modify}(p)) \delta(q \leftarrow P)
\]

\[
\text{ChoiceLB}_{p, q}(\Delta(dd \leftarrow DD(g(i))) \land \text{resp}(\text{locationB}(g(i),dd)) = q)) [\ldots]^{B^M})
\]

\[
\delta(q \leftarrow P) \text{ // Get}_{p, q}(\Delta(i \leftarrow \text{Modify}(p))
\]

\[
\text{ChoiceLB}_{p, q}(\Delta(dd \leftarrow DD(g(i))) \land \text{resp}(\text{locationB}(g(i),dd)) = q)) [\ldots]^{B^M})
\]

In this scheme processor \( p \) is responsible for the computation and \( q \) is responsible for storing the element. The above derived form is hard to comprehend and we therefore introduce the following more intuitive denotation:

\[
\text{ResideB}(i, p) = \{ dd \in DD(g(i)) \mid \text{resp}(\text{locationB}(g(i),dd)) = p \}
\]

This can be expanded for the Load/Send/Receive case:

\[
\Delta(i \leftarrow \text{ModifyA}(\text{mynode})) \text{ChoiceLB}(\Delta(dd \leftarrow \text{ResideB}(i,\text{mynode}))
\]

\[
\text{Load}(\text{mynode}, [\text{locationB}(g(i),dd)]^{B^M})
\]

\[
\delta(q \leftarrow P \setminus \{\text{mynode}\}) \Delta(i \leftarrow \text{ModifyA}(q)) \Delta(dd \leftarrow \text{ResideB}(i,\text{mynode}))
\]

\[
\text{Send}(q, [\text{locationB}(g(i),dd)]^{B^M})
\]

\[
\delta(q \leftarrow P \setminus \{\text{mynode}\}) \Delta(i \leftarrow \text{ModifyA}(\text{mynode}))
\]

\[
\text{ChoiceLB}(\Delta(dd \leftarrow \text{ResideB}(i,q)) \text{Recv}(q, [g(i)]^{B})
\]

**Derivation of Put**

The element \([f(i)]A\) must be stored by the processor(s) responsible for the computation of \([f(i)]A\). As was shown in Section 8.2.1, each processor \( p \) is responsible for computing elements \( \text{ModifyA}(p) \) of \( I \).

\[
\Delta(i \leftarrow \text{ModifyA}(p)) [f(i)]A
\]

We substitute the annotation for \( B \) and we obtain the following \( \text{Get} \) operation:

\[
\delta(p \leftarrow P) \text{ // Put}_{p, q}(\Delta(i \leftarrow \text{ModifyA}(p))
\]

\[
[f(i)] \Delta(q \leftarrow \text{Index}(A)) \Delta(dd \leftarrow DD(q)) [\text{locationA}(q,dd)]^{A^M}
\]

Which after contraction becomes:

\[
\delta(p \leftarrow P) \text{ // Put}_{p, q}(\Delta(i \leftarrow \text{ModifyA}(p)) \Delta(dd \leftarrow DD(f(i)))[\text{locationA}(f(i),dd)]^{A^M}
\]

This last form describes which elements of \( A \) are to be stored by \( p \). Like before, we have to derive which processors are responsible for the memories that store the elements:

\[
\delta(q \leftarrow (P, \text{resp}(\text{locationB}(g(i),dd)) = q))
\]
Substitution in the above Put operation yields:

$$\delta(p \leftarrow P) // \text{Put}_{p}(\Delta(i \leftarrow \text{ModifyA}(p)))
\Delta(dd \leftarrow DD(f(i))) \delta(q \leftarrow (P, \text{resp}(\text{locationB}(g(i), dd)) = q)) [...]A^M)$$

This reduces to:

$$\delta(p \leftarrow P) // \delta(q \leftarrow P) // \text{Put}(\Delta(i \leftarrow \text{ModifyA}(p))) \Delta(dd \leftarrow \text{ResideB}(i,q)) [...]A^M)$$

This can be expanded for the Store/Send/Receive case:

$$\Delta(i \leftarrow \text{ModifyA}(mynode)) \Delta(dd \leftarrow \text{ResideB}(i, mynode))$$
$$\text{Store}(mynode, [\text{locationB}(f(i), dd)]A^M)$$

$$\Delta(i \leftarrow \text{ModifyA}(mynode)) \delta(q \leftarrow P\setminus\text{mynode}) \Delta(dd \leftarrow \text{ResideA}(i,q))$$
$$\text{Send}(q, [\text{locationA}(f(i), dd)]A^M)$$

$$\delta(q \leftarrow P\setminus\text{mynode}) \Delta(i \leftarrow \text{ModifyA}(q)) \Delta(dd \leftarrow \text{ResideA}(i, mynode))$$
$$\text{Recv}(q, [\text{locationA}(f(i), dd)]A^M)$$

Summary

The above demonstrates that the reduction-rules of V-cal can also be used to derive data transports in an automated way. The fact that multiple memories may store the same value and that multiple processors may compute the same value complicates the derivation of Get and Put operations and the associated Load/Store and Send/Receive operations. Moreover, the above scheme is far from optimal, because in the Send and Receive case in the Get operation, each processor where a value resides sends it to the processor(s) that (might) need it. At the receiving end the processor selects one of those messages, and discards the others. The same holds for the Send and Receive case in the Put operation, where multiple processors may compute a certain value and each processor that owns the memory where the value is located will receive duplicates. A more optimal form is to select at the source which message to send.

8.3.2 Data Transport for Views

The derivation of data transport for views is conceptually more complicated than for shapes. In this case, we have a relation between the view identifier V and the shape A that is defined in the algorithm, and a relation between V and its machine image $V^M$ that is defined in the annotation module. Given these relations we must derive the relation between A and $V^M$. We have depicted these relations in Figure 25 on the various definition levels.
The general scheme for the program under consideration is:

\[
\Delta(k \leftarrow K) \bullet (V^k = \sqrt{V(b, \text{True}, dpv(k), ipv(k))A}; \\
W^k = \sqrt{W(b, \text{True}, dpw(k), ipw(k))B}; \\
\Delta(i \leftarrow I) \text{// Clause}\left( [f(i)]V^k \mid [g(i)]W^k \right) )
\]

The general annotation on \(V^k\) and \(W^k\) is equal to:

\[
V^k = \Delta(q \leftarrow J(k)) [\text{location}V(k,q)]V^M \\
W^k = \Delta(q \leftarrow L(k)) [\text{location}W(k,q)]W^M
\]

In the derivation below, we will restricted ourselves to non-overlapping data organisation annotations on views, because this case resembles the overlap annotations on shapes, which was described in the previous section.

Our goal is, given the program on the left, to derive a program of the form on the right:

\[
\begin{align*}
\text{Get} (\Delta(i \leftarrow I) [g(i)]W^k) ; & \quad \text{Get}_p (\Delta(i \leftarrow I(p)) \ldots [\ldots W^M]) ; \\
\Delta(i \leftarrow I) \text{// Clause}( [f(i)]V^k \mid [g(i)]W^k) ; & \quad \Delta(i \leftarrow I(p)) \text{// Clause}( [\ldots V^M \mid [\ldots W^M]) ; \\
\text{Put} (\Delta(i \leftarrow I) [f(i)]V^k) & \quad \text{Put}_p (\Delta(i \leftarrow I(p)) \ldots [\ldots V^M ]) 
\end{align*}
\]

For both the \text{Get} and \text{Put} operations, we have to determine which elements of \(A\) and \(B\) are changed or used, respectively. In order to obtain these sets of required elements and their location, we need to derive an expression of \(A\) in terms of \(V^M\) and \(B\) in terms of \(W^M\). This is given by the following two equations that relate \(V^k\) to \(V^M\) and \(A\), respectively:

\[
\begin{align*}
V^k &= \sqrt{V(b(k), R(k), dp(k), ip(k))A} \\
V^k &= \Delta(q \leftarrow J(k)) [\text{location}V(k,q)]V^M
\end{align*}
\]

where \(I = (a, P) = \text{Index}(A)\) and \(J(k) = (b \& dp(k, a), P(ip(k)) \land R(k))\). To derive the relation between \(A\) and \(V^M\), we have to find an expression of \(A\) in terms of \(V^k\). In Chapter 6, an expression of an shape in terms of its view was given
(Theorem V20), but (multiple) elements of $V^k$ may correspond with an element of $A$. We must therefore choose:

$$[i]A = \text{ChoiceLA}(\Delta(t \leftarrow (J(k) \mid ip(k,t) = i)) [t] V^k)$$

(3)

where $I = (\alpha, P) = \text{Index}(A)$ and $J(k) = (b \& dp(k, \alpha), P(ip(k)) \land R(k))$. We substitute (3) in (2), and obtain expression (4) after the application of contraction (V10b):

$$[i]A = \text{ChoiceLA}(\Delta(t \leftarrow (J(k) \mid ip(k,t) = i)) [t] \Delta(q \leftarrow J(k)) [\text{location} V(k, q)] V^M)$$

$$\Leftrightarrow [i]A = \text{ChoiceLA}(\Delta(t \leftarrow (J(k) \mid ip(k,t) = i)) [\text{location} V(k, t)] V^M)$$

(4)

We have now obtained a view expression that defines where the element $[i]A$ is to be stored for each $k$. This result is again substituted in the scheme of elements to be retrieved:

$$\Delta(i \leftarrow \text{Modify}(k,p)) [f(i)] V^k = \Delta(i \leftarrow I) [f(i)] \forall V(b(k), R(k), dp(k), ip(k)) A$$

yielding:

$$\Delta(i \leftarrow \text{Modify}(k,p)) [f(i)] \forall V(b(k), R(k), dp(k), ip(k)) = \Delta(i \leftarrow \text{Modify}(k,p))$$

$$\Delta(i \leftarrow \text{Modify}(k,p)) [f(i)] \forall V(b(k), R(k), dp(k), ip(k)) \Delta(s \leftarrow I)$$

$$\text{ChoiceLA}(\Delta(t \leftarrow (J(k) \mid ip(k,t) = s)) [\text{location} V(k, t)] V^M) =$$

$$\Delta(i \leftarrow \text{Modify}(k,p))$$

$$\text{ChoiceLA}(\Delta(t \leftarrow (J(k) \mid ip(k,t) = ip(k,f(i)))) [\text{location} V(k, t)] V^M)$$

For convenience, the following short-hand notation is introduced:

$$\text{Elements}(k,i,p) = \Delta(t \leftarrow (J(k) \mid ip(k,t) = ip(k,f(i)))) [\text{location} V(k, t)] V^M$$

The scheme for the Get and Put operations equals:

$$\delta(p \leftarrow P) /\Delta(k \leftarrow K) \ast \{$$

$$\text{Get}_p(\Delta(i \leftarrow \text{Modify}(k,p)) \text{ChoiceLA} (\text{Elements}(k,i,p)) );$$

$$\Delta(i \leftarrow \text{Modify}(k,p)) /\Delta(\text{Clause}([ip_V(f(i)])A \mid [ip_W(g(i)])B );$$

$$\text{Put}_p(\Delta(i \leftarrow \text{Modify}(k,p)) \text{Elements}(k,i,p)) ) \}$$

(5)

The expansion to the load/store/send/receive type of data transports is performed in the same way as defined in the section of data organisation annotations on shapes. We therefore restrict ourselves to the Get and Put scheme.

Example 8.5

Consider a view on $A$ with a (composed) index propagation function: $ip_V(i,j) = i+j$ and a domain for $(i,j) \in J(k) = (0:n-k \times 0:n-k)$. The expression for Elements than equals:
Elements\((k,i,p) = \Delta(t \leftarrow ((0:n-k \times 0:n-k) \mid t_1+t_2 = f(i)_1+f(i)_2)) \) \(\text{[location}\(V(k,t)\)]V^M\)

The equation \(ip_V(k,t) = ip_V(k,f(i))\) or \(t_1 + t_2 = f(i)_1 + f(i)_2\) has various solutions. The equation \(t_1 + t_2 = f(i)_1 + f(i)_2\) has a range of solutions for a given \(f(i)\), namely \(t_1 + t_2 = constant\), where \((t_1, t_2) \in (0:n-k \times 0:n-k)\).

**Optimisation**

Version (7) of the \(Get/\)\(Put\) translation can be simplified if equation \(ip_W(k,t) = ip_W(k,g(i))\) is resolved and a closed form expression is obtained for the range of \(t\) in terms of \(i\). This equation can be resolved iff the inverse for \(ip_V\) and \(ip_W\) exists and, in addition, \(ip_V^{-1}\) and \(ip_W^{-1}\) are total functions. The form then reduces to the following \(V-cal\) program, where \(ip_V^{-1}(k,ip_V(k,f(i))) = f(i)\):

\[
\delta(p \leftarrow P) / \Delta(k \leftarrow K) \cdot \{
\begin{align*}
&\text{Get}(\Delta(i \leftarrow \text{Modify}(k,p)) \ [\text{location}\(W(k,f(i))\)]W^M)));
&\Delta(i \leftarrow \text{Modify}(k,p)) / \text{Clause}([ip_V(f(i))]A \mid [ip_W(g(i))]B);
&\text{Put}(\Delta(i \leftarrow \text{Modify}(k,p)) \ [\text{location}\(V(k,f(i))\)]V^M))
\end{align*}
\] (8)

**Summary**

The integration of data organisation annotations on views is somewhat complicated, because these type of annotations are "virtual" and have to be related to the shape they are defined on. When a view is not a function, the annotation on the view may result in a duplication of the elements of the shape this view refers to. Hence, apart from data duplication definitions in annotations on shapes, we may realise data duplications by annotating an overlapping view. An example of this is found for the \(Red/Black\) algorithm described in Section 4.6.2 and annotated in Section 5.5.2.

**8.4 Index set Optimisations**

The \((parallel)\) \(V-cal\) programs presented above all include the computation of sets like:

\[\text{Set}(p) = \{ i \in \mathbb{N}^b \mid \text{proc}\(V(k,f(i)) = p\)\}\]

When applying the index overhead measure \(IM\) that was introduced in Chapter 7 we obtain in the general case \(IM(\text{Set}(p)) = |b|/(|b|/p_{\text{max}}) = p_{\text{max}}\). It is evident that the compile-time conversion of these sets, such that \(IM\) is minimal, is important for run-time overhead reduction. However, this overhead measure cannot always be determined at compile-time.

First, the predicate of the index set may depend on the content of some data structure (which is only known at run-time) and a reduction at compile-time is only partly possible. Secondly, an optimal index set may exist, but the automatic reduction of these forms is too complicated. The latter restriction more or less
depends on the progress made in — automated — symbolic manipulation, whereas the first one is inherent to the program. The only optimisation achievable here is the recognition and reduction of general patterns.

Below, we will restrict ourselves to the various optimisations possible at compile-time, where we will only consider the proc function, because optimisation of the location function is comparable. We will restrict ourselves to non-overlapping proc functions.

### 8.4.1 Optimisation scheme

The best achievable solution to the membership problem, at compile-time, is to find a generation function gen the predicate proc\(_A(f(i)) = p\). The function gen \(\subseteq (i_{\min} : i_{\max}) \times (t_{p,\min} : t_{p,\max})\) is to be constructed such that \(i = gen(t)\) and the predicate proc\(_A(f(i)) = p\) is always True in the range \((t_{p,\min} : t_{p,\max})\), effectively replacing:

\[
A = \Delta(p \leftarrow (0 : p_{\max} - 1)) \Delta(i \leftarrow (i_{\min} : i_{\max} \mid \text{proc}_A(f(i)) = p)) [p, \text{location}_A(f(i))]A'
\]

by

\[
A = \Delta(p \leftarrow (0 : p_{\max} - 1)) \Delta(t \leftarrow (t_{p,\min} : t_{p,\max})) [\text{gen}_p(t)]
\hspace{1em}
\Delta(i \leftarrow (i_{\min} : i_{\max} \mid \text{proc}_A(f(i)) = p)) [p, \text{location}_A(f(i))]A'
\]

yielding:

\[
A = \Delta(p \leftarrow (0 : p_{\max} - 1)) \Delta(t \leftarrow (t_{p,\min} : t_{p,\max})) [p, \text{location}_A(f(\text{gen}_p(t)))]A'
\]

The function gen generates exactly those indices that are effectively used. For a number of computation organisations and properties of the function \(f(i)\) those optimisations can be obtained. We will elaborate on those optimisations in the following sections, where we will concentrate on the class of well known types of Block-Scatter computation partitioning: BS(b) with a certain block-size b. Prior to this, we present a general special case, namely if the index propagation function \(f\) is a constant.

**Theorem 01:**

Let \(A\) be subject to any computation organisation, and let \(f(i) = b\), where \(b\) is an integer constant, then the generation parameters are given by:

\[
\begin{align*}
gen(t) &= t \\
t_{p,\min} &= i_{\min}, & \text{for } p = \text{proc}_A(b) \text{ and } t_{p,\min} = 0, \text{ for } p \neq \text{proc}_A(b) \\
t_{p,\max} &= i_{\max} & \text{for } p = \text{proc}_A(b) \text{ and } t_{p,\max} = -1, \text{ for } p \neq \text{proc}_A(b)
\end{align*}
\]

**Proof:**

The condition proc\(_A(f(i)) = p\) implies proc\(_A(b) = p\), which implies that the responsibility assignment is independent of \(i\). Consequently, for processor \(p = \text{proc}_A(b)\), the complete range \(i_{\min} : i_{\max}\) is valid and for all other proces-
sors the range is empty. For \( \text{gen}(t) \) we can take any arbitrary function, because it is composed with the constant \( b \), that yields again a constant function.

\[ \square \]

### 8.4.2 Block-Scatter Partitionings

In block-scatter partitioning the data is split in a number of blocks containing consecutive data elements, each block scattered over the range of processors. The optimisation is given in the following theorem:

**Theorem 8.2 Block-scatter partitionings**

Let \( A' \) be a block-scatter-partitioning \( \text{BS}(b) \) of \( A \) with parameter \( b \) according to:

\[
A = \Delta(i \leftarrow (i_{\min}; i_{\max})) \Delta(j \leftarrow (j_{k,p,\min}; j_{k,p,\max})) [p, \text{location}(f(i)))] A',
\]

where \( \text{proc}(i) = (i \ mod \ b) \ mod \ p_{\max} \) and \( \text{location}(i) = (i \ mod \ b) \ div \ p_{\max} \). Let \( f \) be a monotonic increasing function, then:

\[
A = \Delta(k \leftarrow (0; k_{p,\max}))) \Delta(j \leftarrow (j_{k,p,\min}; j_{k,p,\max})) [p, \text{location}(f(j)))] A', \text{ where}
\]

\[
k_{p,\max} = f(i_{\max}) \ div \ b - p \ div \ p_{\max}
\]

\[
j_{p,k,\min} = \max(i_{\min}, \lceil f^1(b^*(p + k^*p_{\max})) \rceil)
\]

\[
j_{p,k,\max} = \min(i_{\max}, \lceil f^1(b^*(p + k^*p_{\max}) + b - 1) \rceil)
\]

**Proof:**

We can write the equation \( p = (f(i) \ div \ b) \ mod \ p_{\max} \) also as:

\[
p + k^*p_{\max} = t \ \text{with} \ t = f(i) \ div \ b
\]

Let the range for \( k \) be equal to \( 0:k_{\max} \) and assume that \( f \) is a monotonic increasing function, then the value for \( k_{\max} \) can be derived as follows:

\[
k_{\max} = \max( k \in N \ \mid p + k^*p_{\max} \leq f(i_{\max}) \ div \ b ) = \max( k \in N \ \mid k \leq (f(i_{\max}) \ div \ b - p) \ div \ p_{\max} ) = (f(i_{\max}) \ div \ b - p) \ div \ p_{\max}
\]

The range for \( j_{p,k} \) in \( f(j_{p,k}) \ div \ b = p + k^*p_{\max} \) can be derived as follows:

\[
b^*(p + k^*p_{\max}) \leq f(j_{p,k}) \leq b^*(p + k^*p_{\max}) + b - 1 \ \Rightarrow \ [f^1(b^*(p + k^*p_{\max}))] \leq j_{p,k} \leq [f^1(b^*(p + k^*p_{\max}) + b - 1)]
\]

The range thus equals:

\[
j_{p,k,\min} = \max(i_{\min}, [f^1(b^*(p + k^*p_{\max}))])
\]

\[
j_{p,k,\max} = \min(i_{\max}, [f^1(b^*(p + k^*p_{\max}) + b - 1)])
\]

In those cases were \( j_{p,k,\min} > j_{p,k,\max} \), there exists no integer solution for the range of \( j_{p,k} \).
4

\textbf{i. Repeated Scatter}

The formula found for the BS(b) can be rewritten by applying rules from the calculus, to obtain a form that is under the condition that \( b \geq f(i_{\text{max}})/(2p_{\text{max}}) \) more favourable than the form in the previous Theorem. We will refer to this form as the \textit{Repeated Scatter partitioning} as opposed to the \textit{Repeated Block partitioning} that was presented in the above Theorem.

\[ A = \Delta(k \leftarrow ((0:k_{\text{p, max}})) \Delta(j \leftarrow (j_{k,p,\text{min}}:j_{k,p,\text{max}})) [p, \text{location}(f(j))] A' = \]
\[ = \Delta(k \leftarrow ((0:k_{\text{p, max}})) \Delta(t \leftarrow (b-p:b+b-1)) \]
\[ \Delta(j \leftarrow (j_{k,p,\text{min}}(t):j_{k,p,\text{max}}(t))) [p, \text{location}(f(j))] A', \text{ where} \]
\[ j_{p,k,\text{min}}(t) = \max(i_{\text{min}}, \lfloor f^{-1}(t + b*k_{\text{p, max}}) \rfloor) \]
\[ j_{p,k,\text{max}}(t) = \min(i_{\text{max}}, \lfloor f^{-1}(t + b*k_{\text{p, max}}) \rfloor) \]
\[ = \Delta(t \leftarrow (b-p:b+b-1)) \Delta(k \leftarrow ((0:k_{\text{p, max}}) \mid t + b*k_{\text{p, max}} \in \mathbb{Z}) \]
\[ [p, \text{location}(t + b*k_{\text{p, max}})] A' \]

In the following, we will explore a number of special cases of block-scatter partitionings, such as block partitioning and scatter partitioning in a general form and with a specific form of the index propagation function \( f \).

\textbf{ii. Block-partitioning}

In the case of block-partitioning, the general case of block-scatter partitionings reduces to BS(b), where \( b = f(i_{\text{max}})/p_{\text{max}} \leftrightarrow p_{\text{max}} * b = f(i_{\text{max}}) \). From this it follows that \( k_{\text{p, max}} = (f(i_{\text{max}}) \div b - p) \div p_{\text{max}} = (p_{\text{max}} \div b) \div p_{\text{max}} = (p_{\text{max}} - p) \div p_{\text{max}} = 0 \), and the parameter \( k \) can be eliminated altogether. The bounds for \( j \) now equal:

\[ j_{p,\text{min}} = \max(i_{\text{min}}, \lfloor f^{-1}(b*p) \rfloor) \]
\[ j_{p,\text{max}} = \min(i_{\text{max}}, \lfloor f^{-1}(b*p+b-1) \rfloor) \]

The theorems above are also valid for monotonic decreasing functions \( f(i) \), provided that the arguments of \( f^{-1} \) are exchanged for \( t_{p,\text{min}} \) and \( t_{p,\text{max}} \). The functions \( \max \) and \( \min \) can only be eliminated if \( i_{\text{min}}, i_{\text{max}}, \) and \( p \) are known at compile-time, otherwise they remain as a function in the SPMD-code.

\textbf{iii. Scatter-partitioning}

In the case of scatter-partitioning the general case of Block-scatter partitionings reduces to BS(1) which yields the following simplified ranges:

\[ k_{\text{p, max}} = (f(i_{\text{max}}) - p) \div p_{\text{max}} \]
\[ j_{p,k,\text{min}} = \max(i_{\text{min}}, \lfloor f^{-1}(p + k*p_{\text{max}}) \rfloor) \]
\[ j_{p,k,\text{max}} = \min(i_{\text{max}}, \lfloor f^{-1}(p + k*p_{\text{max}}) \rfloor) \]
The next theorem defines a more optimal version for the scatter decomposition if the function $f$ is linear:

**Theorem 8.3 Scatter-partitioning with linear function**

Consider the scatter partitioning BS(1) with function $f(i) = a \cdot i + c$, $a \in Z \setminus \{0\}$, $c \in Z$, then the view expression can be rewritten as:

$$A = \Delta(t \leftarrow (t_{p, min}, t_{p, max})) \ [proc(f(g_{p}(t))), location(f(g_{p}(t)))) \ A']$$

$$gen_{p}(t) = x_{p} + (p_{max}/gcd(a,p_{max})) t$$

$$t_{p, min} = [(i_{min} - x_{p})/(p_{max}/gcd(a,p_{max}))]$$

$$t_{p, max} = [(i_{max} - x_{p})/(p_{max}/gcd(a,p_{max}))]$$

where $x_{p}$ is a particular solution in $i$ of the linear diophantine equation $a \cdot i - p_{max} \cdot k = p - b$ and $gcd$ denotes the function that yields the greatest common divisor. If no solution to the diophantine equation exists, then that particular processor is not to execute any code.

**Proof:**

The index propagation function has the form $f(i) = a \cdot i + c$, $a \in Z \setminus \{0\}$, $c \in Z$, and the inverse exists and equals $f^{-1}(i) = (i - c)/a$. Hence by Theorem 2, we have the following values for the bounds $j_{p, k, min}, j_{p, k, max}$ and $k_{max}$:

$$k_{max} = (a \cdot i_{max} + c - p) \ div \ p_{max}$$

$$j_{p, k, min} = \max(i_{min}, \ceil{f^{-1}(p + k \cdot p_{max})}) = \max(i_{min}, \ceil{(p + k \cdot p_{max}) - c}/a))$$

$$j_{p, k, max} = \min(i_{max}, \floor{f^{-1}(p + k \cdot p_{max})}) = \min(i_{max}, \floor{(p + k \cdot p_{max}) - c}/a))$$

Thus $j_{p, k, min} = j_{p, k, max}$ iff $(p + k \cdot p_{max}) - c/a = i \in Z$. This condition for $k$ can be restated as:

$$(p - c) = a \cdot i - k \cdot p_{max} \quad \text{where} \quad 0 \leq k \leq k_{max}$$

Let $i = x_{p}$ be a particular solution of this diophantine equation, which is obtained by the usual iterative technique of logarithmic complexity. Consequently the generation function $gen_{p}$ is given by $gen_{p}(t) = x_{p} + (p_{max}/gcd(a,p_{max})) t$, iff $(-p + c)/gcd(a,p_{max}) \in Z$. The range $t_{p, min} \leq t \leq t_{p, max}$ can be derived from the constraint $i_{min} \leq gen_{p}(t) \leq i_{max}$. Substitution of $gen_{p}(t)$ yields $i_{min} \leq x_{p} + (p_{max}/gcd(a,p_{max})) t \leq i_{max}$ and the corresponding ranges are:

$$t_{p, min} = \floor{(i_{min} - x_{p})/(p_{max}/gcd(a,p_{max}))}$$

$$t_{p, max} = \ceil{(i_{max} - x_{p})/(p_{max}/gcd(a,p_{max}))}$$
### Table I. Optimisations for several partitionings. RB means repeated block-partitioning and RS means repeated scatter partitioning.

<table>
<thead>
<tr>
<th>f(i)/g(i)</th>
<th>Block</th>
<th>Scatter</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>( \text{gen}<em>P(t) = t ) ( t</em>{p,\min} = i_{\min}, \text{for } p = c \text{ div } b ) ( t_{p,\max} = i_{\max}, \text{for } p = c \text{ div } b ) (else ( t_{p,\min} = 0 ) and ( t_{p,\max} = -1 ))</td>
<td>( \text{gen}<em>P(t) = t ) ( t</em>{p,\min} = i_{\min}, \text{for } p = c \text{ mod } p_{\max} ) ( t_{p,\max} = i_{\max}, \text{for } p = c \text{ mod } p_{\max} ) (else ( t_{p,\min} = 0 ) and ( t_{p,\max} = -1 ))</td>
</tr>
<tr>
<td>i+c</td>
<td>( \text{gen}<em>P(t) = t ) ( t</em>{p,\min} = \text{Max}(i_{\min}, \lceil b \cdot p \cdot c \rceil) ) ( t_{p,\max} = \text{Min}(i_{\max}, \lfloor b \cdot p + b - 1 - c \rfloor) )</td>
<td>( \text{gen}<em>P(t) = p \cdot c + (p</em>{\max} \cdot t) ) ( t_{p,\min} = \lceil (i_{\min} \cdot p + c) / p_{\max} \rceil ) ( t_{p,\max} = \lfloor (i_{\max} - p + c) / p_{\max} \rfloor )</td>
</tr>
<tr>
<td>a*i+c</td>
<td>see ( a \cdot i + c ) ( p_{\max} \mod a = 0 )</td>
<td>( \text{gen}<em>P(t) = (p \cdot c + p</em>{\max} \cdot t) / a ) ( t_{p,\min} = \lceil (a \cdot i_{\min} \cdot p + c) / p_{\max} \rceil ) ( t_{p,\max} = \lfloor (a \cdot i_{\max} \cdot p + c) / p_{\max} \rfloor )</td>
</tr>
<tr>
<td>a*i+c</td>
<td>see ( a \cdot i + c ) ( a \mod p_{\max} = 0 )</td>
<td>( \text{gen}<em>P(t) = t ) ( t</em>{p,\min} = i_{\min}, \text{for } p = c \text{ mod } p_{\max} ) ( t_{p,\max} = i_{\max}, \text{for } p = c \text{ mod } p_{\max} ) (else ( t_{p,\min} = 0 ) and ( t_{p,\max} = -1 ))</td>
</tr>
<tr>
<td>a*i+c</td>
<td>( \text{gen}<em>P(t) = t ) ( t</em>{p,\min} = \text{Max}(i_{\min}, \lceil (b \cdot p - b) / a \rceil) ) ( t_{p,\max} = \text{Min}(i_{\max}, \lfloor (b \cdot p + b - 1 - c) / a \rfloor) )</td>
<td>( \text{gen}<em>P(t) = x</em>{p} + (p_{\max} / \gcd(a, p_{\max}) \cdot t) ) ( t_{p,\min} = \lceil (i_{\min} \cdot x_{p}) / (p_{\max} / \gcd(a, p_{\max}) \rceil ) ( t_{p,\max} = \lfloor (i_{\max} - x_{p}) / (p_{\max} / \gcd(a, p_{\max}) \rfloor )</td>
</tr>
<tr>
<td>monotonic increasing</td>
<td>( \text{gen}<em>P(t) = t ) ( t</em>{p,\min} = \text{Max}(i_{\min}, \lfloor f^{-1}(b \cdot p) \rfloor) ) ( t_{p,\max} = \text{Min}(i_{\max}, \lceil f^{-1}(b \cdot p + b - 1) \rceil) )</td>
<td>no optimisation</td>
</tr>
</tbody>
</table>
8.5 Data transport Optimisations

The membership test optimisations discussed above reduce the — unnecessary — overhead in generating those indices involved in local computations on a given processor. Data transport-patterns to a great extent determine the overall performance of a parallel program on for example a distributed-memory architecture. Some patterns may lead to idling processors or to link-contention.

The combination of algorithm and annotation specification as described by the programmer determines the data transport-pattern. From these specifications, the compiler can derive a variety of patterns that satisfy the specifications, but with a different behaviour when executed on a specific architecture. Obviously, the compiler should select the most favourable option for a given architecture based upon some performance model.

We conceive a number of optimisations that are all primarily induced by the limitations of the interconnection network and communication software. Among these are problems like link-contention, message-buffer overflow, data transport-lateness, and inefficiencies caused by the discrepancy between the software- and hardware communication structure. Several general optimisations are discussed below.

8.5.1 Optimisation Metric

An important factor for overall performance of a program on a machine is the overhead caused by data transport induced by the given data and computation organisation definition. We therefore define a measure on programs that returns an estimate for the total amount of data transport. The measure for data transport is especially useful as a comparison between various annotations. For the definition of the measure, we distinguish between three types of data transport. The first type is the initial data transport for loading data from secondary memory to the appropriate memories according to the data organisation definition. The second type is data transport induced during the execution phase of the program. The third type is the final data transport phase: the transport of all output data to secondary memory. Below, we present the definition for the second data transport measure:

Definition 8.4

Let $\text{DTM} : \text{pow( V-cal-programs )} \rightarrow N$ and $\text{Op}_1 \in \{\text{Store, Load, Put, Get}\}$ and $\text{Op}_2 \in \{\text{Send, Recv}\}$:

$$\text{DTM}(\Delta(k \leftarrow K \text{Op}_1(ve(k)))) = 1 \text{ Index} (\Delta(k \leftarrow K) ve(k)),$$

$$\text{DTM}(\Delta(k \leftarrow K \text{Op}_2(ve(k)))) = 2^* 1 \text{ Index} (\Delta(k \leftarrow K) ve(k)),$$

$\text{DTM( Clause )} = 0$, iff $\text{Clause}$ does not contain $\text{Op}$.

$\text{DTM( Clause}_1 \oplus \text{Clause}_2) = \text{DTM( Clause}_1) + \text{DTM( Clause}_2)$
Consider the following example to assess the use of the DTM measure. Here VM is an virtual machine with $p$ processors and $p$ associated memories.

\[
\begin{align*}
\text{SHAPE } &A, B \ (n \times n) \text{ OF REAL;} \\
&&\text{ALTERNATIVE} \\
& A := 6B + 1; \\
& A \leftarrow \text{Cyclic(VM)}; B \leftarrow \text{Block(VM)}; \\
\text{OR} \\
& A \leftarrow \text{Block(VM)}; B \leftarrow \text{Block(VM)}; \\
\text{END}
\end{align*}
\]

For the first alternative DTM equals $2n^2$ for the load phase, $3n^2$ for the computation and $n^2$ for the store phase. For the second alternative the DTM equals $2(n \ \text{div} \ p)$, that is only the communication across the border of each of the blocks of $A$ for the second content statement. If in addition the real machine has a message-passing programming system, the number of operations for the DTM have to be multiplied by two, because two operations: a send and a receive are needed for each Get and Put.

The above example shows that the Data Transport overhead Measure can be used to quantify the amount of communication in the program. This can again be used to statically compare various annotation strategies.

### 8.5.2 Data Transport Elimination and Re-ordering

The clause by clause translation discussed in this chapter results in a sequence of Get Clause Put patterns. This pattern may not be optimal, because it may contain patterns that store elements into memory and retrieve it in the next operation. Moreover, some machines have interconnection networks with long delays and it may be advantageous to execute the operation that fetches data-items at an earlier stage.

**Prefetching and Elimination of Data Transport**

If no data dependencies exist, the Get and Put operations can be interchanged. The same holds for delaying Put operations. The effect of prefetching largely depends on the facilities offered by the real machine to exploit them, such as potential overlap in computation and data transport operations and the availability of buffer space.

The second type of optimisation implements a type of “compile-time” caching strategy. If data-elements are computed in one clause and needed in a later stage they could be stored immediately and retrieved later, but they could also be stored near that processor until the succeeding clause has used it.
might be of interest for those real machines that have long communication delays and small bandwidth.\textsuperscript{3}

The principle of pre-fetching is illustrated in Figure 26:

```
Get(X);
Clause 1 (Y | X);
Put(Y);
---
Get(X);
Get(Z);
Clause 1 (Y | X);
Put(X);
---
Get(X);
Get(Z);
Clause 2 (X | X, Z);
Put(X);
---
Get(X);
Get(Z);
Clause 2 (X | X, Z);
Put(Y);
```

\[ Figure \ 26 \  The \ principle \ of \ pre-fetching \ and \ elimination \]

Note that the DTM, when the sizes of \( X, Y, \) and \( Z \) are equal to one, changes from 5 to 4 in the above scheme. The reduction-rule underlying these transformations is the following:

**Definition O4 Data Transports Interchange and Elimination**

Let \( ve_1 \) and \( ve_2 \) be view expressions and \( # \in \{ \text{Put}, \text{Get} \} \), than if

\[
\begin{align*}
\text{Ref}(ve_1) \cap \text{Ref}(ve_2) &= \emptyset \text{ then } #_1(ve_1); #_2(ve_2) \Leftrightarrow #_2(ve_1); #_1(ve_2) \\
\text{Ref}(ve_1) \cap \text{Ref}(ve_2) &\neq \emptyset \text{ then } \\
\text{Put}(ve_1); \text{Get}(ve_2) &\Leftrightarrow \text{Put}(\text{Ref}(ve_1) \setminus \text{Ref}(ve_2)); \text{Get}(\text{Ref}(ve_2) \setminus \text{Ref}(ve_1)) \\
\text{Get}(ve_1); \text{Put}(ve_2) &\Leftrightarrow \text{Get}(\text{Ref}(ve_1) \setminus \text{Ref}(ve_2)); \text{Put}(\text{Ref}(ve_2) \setminus \text{Ref}(ve_1)) \\
#(ve_1); #(ve_2) &\Leftrightarrow #(\text{Ref}(ve_1) \cup \text{Ref}(ve_2))
\end{align*}
\]

\[ \square \]

**Non-overlapping Annotations**

The Booster annotation language allows for the definition of overlapping data- and computation organisations. The schemes that have been presented in this chapter take these type of organisations into account. If, however, the annotations are non-overlapping this general scheme is not the most efficient one. We therefore present for a number of cases a more optimal scheme. First, we introduce a number of sets that can be used to classify the type of annotation.

Given the annotation of a shape \( A \), we define the \( \text{DataDup} \) set, which returns the number of times a certain data item has been duplicated:

\[
A = \Delta(i \leftarrow I) \text{ChoiceLA}(\Delta(\mathbf{dd} \leftarrow DD(i)) [\text{locationA}(i, \mathbf{dd})])^{A^M}) \\
\text{DataDup}(X) = \{ i \leftarrow I \mid \sum_i DD(i) \}
\]

\[ \text{Note that this optimisation is bounded by the storage resources available locally, and cannot be applied infinitely.} \]
The same holds for the CompDup set, which returns the number of times a certain computation responsibility for a certain data item has been duplicated:

\[ A = \Delta(i \leftarrow I) \text{ChoicePA} (\Delta(p_d \leftarrow PD(i)) [\text{procA}(i,p_d), \ldots] A^P) \]

\[ \text{CompDup}(X) = \{ i \leftarrow I | \sum_i PD(i) \} \]

When \(|\text{DataDup}(X)\| = |X|\) and \(|\text{CompDup}(X)\| = |X|\), we have a non-overlapping data organisation annotation. The ChoicePA and ChoiceLA functions used in the annotations can be replaced by identity functions. The schemes for the Get and Put operations can now be simplified. The Get operation becomes:

\[ \text{Modify}_X(p) = \{ i \in I | \text{proc}_X(f(i)) = p \} \]

\[ \text{Reside}_Y(p) = \{ i \in I | \text{resp}(\text{location}_Y(g(i))) = p \} \]

\[ \Delta(i \leftarrow \text{ModifyA}(p) \cap \text{ResideB}(p)) \quad \text{Load}(p, [\text{location}_B(g(i))]) B^M \]

\[ \Delta(i \leftarrow \text{ResideB}(p) \setminus \text{ModifyA}(p)) \quad \text{Send}(p, [\text{location}_B(g(i))]) B^M \]

\[ \Delta(i \leftarrow \text{ModifyA}(p) \setminus \text{ResideB}(p)) \quad \text{Recv}(\text{resp}(\text{location}_B(g(i))), [\text{location}_B(g(i))]) B^M \]

The Put operation becomes:

\[ \Delta(i \leftarrow \text{ModifyA}(p) \cap \text{ResideA}(p)) \quad \text{Store}(p, [\text{location}_A(f(i))]) A^M \]

\[ \Delta(i \leftarrow \text{ModifyA}(p) \setminus \text{ResideA}(p)) \quad \text{Send}(\text{resp}(\text{location}_A(g(i))), [\text{location}_A(f(i))]) A^M \]

\[ \Delta(i \leftarrow \text{ResideA}(p) \setminus \text{ModifyA}(p)) \quad \text{Recv}(\text{procA}(f(i)), [f(i)]) A \]

### 8.5.3 Index Sets and Data Transport

The relations that define processor and memory responsibility, proc and mem, and the function location, that specifies the ordering within a certain memory, may depend on the content of some data structure. Likewise, the views \( V = (K,dp,ip) \) and \( W = (K,dp,ip) \) can depend on the content of the data structures they are applied to. An example of this is the following annotated Booster program:

```plaintext
MODULE Comp (C, A) -> (B) |
C, A (n) OF INT; |
BEGIN |
\quad B[A > 0] := C[A > 0] * 5 |
END. |
VIRTUAL MACHINE vm (PROC proces(p)); |
IMPORT Comp:: A (m) |
\quad A[i] <- proces[i \div (m \div p)]; |
```

For these type of views and partitionings, we can not derive the kind of compile-time reductions as presented in the previous section. An option is to make use of the structure present in these cases to reduce the resulting run-time
overhead. In addition, we can choose between several approaches to solve the induced run-time data transport overhead. The best approach depends on the type of architecture.

**General case:**
We now consider the following general form of a data-dependent index set in a parameter binding, where the structure on which it depends (in our template \(A\)) is annotated. We distinguish between data-dependent parameter bindings in view expressions and clauses.

\[
\Delta(i \leftarrow (a, R(A))) \text{ ViewExpression}(i) \quad \text{and} \quad
\Delta(i \leftarrow (a, R(A, i))) \triangleq \text{Clause}(i)
\]

We now introduce the partitioning of \(A\):

\[
A = \Delta(q \leftarrow I) \{ \text{procA}(q), \text{locationA}(q) \} A'
\]

When we substitute the annotation of \(A\) in a data dependent view expression, we obtain an index set that cannot be determined at compile-time and we therefore need to compute the index set at run-time before it is used in the view expression. Below, we will use the notation \(I := (a, R(A))\) to denote the computation of an index set. In this case, \(I\) is a new data structure that stores the elements of the index set \(I\). The problem of computing these sets was already discussed in Section 7.2 on the translation of the *Re number* view.

\[
\{ I := (a, R(A)) \}
\]

\[\quad\]

\[
\Delta(i \leftarrow I) \text{ ViewExpression}(i)
\]

The need to compute the index set at run-time also holds for clauses. Immediately preceding the clause the index set is computed:

\[
\Delta(i \leftarrow (a, R(A))) \triangleq \text{Clause}(i) \equiv \{ I := (a, R(A)) \} \circ \Delta(i \leftarrow I) \triangleq \text{Clause}(i)
\]

We select the following three options for the computation of this intermediate value \(I\), which we will express in terms of an annotation on \(I\):

1. **Single-processor computation**
   The value for \(I\) is computed on one *single* processor by first accumulating the values of \(A\) followed by the distribution of the results. In terms of annotations, this is formalised as follows, where computation on processor 0 is assumed:

   \[
   I = \Delta(q \leftarrow Q) [0, q] I'
   \]
2. All-processor computation
The value for \( I \) is computed on every processor. In terms of annotations
this is a computation duplication:

\[
I = \Delta(q \leftarrow Q) \Delta(t \leftarrow (0:p_{\text{max}})) [t,q] I'
\]

3. Distributed computation
The last option is to compute parts of \( I \) on the same processor where \( A \)
resides, and exchange results:

\[
I = \Delta(q \leftarrow Q) [\text{proc} A(q), \text{location} A(q)] I'
\]

Example 8.6
The evaluation of, for example, an index propagation function that depends
on the content of a distributed shape cannot be performed locally on a single
processor, but information from other processors is needed. Below, we illus-
trate the translation of these kind of functions with the selection of the pivot
in the second Gaussian elimination example.

\[
I := (a,R(V)) = (i \in N^a \mid \{ i = \text{index(reduce(maxindex, } \Delta(x \leftarrow I)) \text{ abs([x]V))} \})
\]

This function selects those elements of some vector \( V \) for which the absolute
value of the content is maximal and selects the first element if more than one
element satisfies the condition.

To compute \( I \), we select the third option. We assume that vector \( V \) has been
annotated and from this we can deduce the computation of the index set by
replacing \( V \) by its annotation:

\[
R(V,i) = \{ i = \text{index(reduce(maxindex, } \Delta(x \leftarrow I)) \text{ abs([proc}V(x), \text{location}(x)]VM)) \} \Rightarrow \text{Extraction}
\]

\[
\{ i = \text{index(reduce(maxindex, } \Delta(x \leftarrow I)) \text{ } \delta(p \leftarrow ((0:p_{\text{max}}-1), \text{proc}V(x) = p)) \text{ abs([}p, \text{location}(x)] VM) \} \Rightarrow V16
\]

\[
\{ i = \text{index(reduce(maxindex, } \delta(p \leftarrow (0:p_{\text{max}}-1)) \text{ reduce(maxindex, } \Delta(x \leftarrow (a,P \land \text{proc}V(x) = p)) \text{ abs([}p, \text{location}(x)] VM) \} \}
\]

The resulting predicate describes a version of the index propagation function
where each processor computes the maximal index for parts of \( VM \) for which
that processor is responsible, and subsequently these results are accumulated
in a second reduction. The introduction of data transport leads to the following
scheme:

\[
I := (a,i = \text{index(reduce(maxindex, } \delta(p \leftarrow (0:p_{\text{max}})) \text{ reduce(maxindex, } \Delta(x \leftarrow (a,P \land \text{proc}V(x) = p)) \text{ abs([}p, \text{location}(x)] VM) )};
\]

\[
\Rightarrow
\]
\[ T := \text{index}(\text{reduce}(\text{maxindex}, \Delta(x \leftarrow (\alpha, p \land (\text{proc}V(x) = p))) \text{abs}([p, \text{location}(x)] VM)) ); \]
\[ \text{pivot} := \text{index}(\text{reduce}(\text{maxindex}, \delta(p \leftarrow (0:p_{\text{max}})) [p]T)) \]

We now associate a central server annotation with \text{pivot}:
\[ \text{pivot} = \Delta(t \leftarrow 0) [t,0] \text{pivot}^p \]

A scheme with data transport then equals:
\[
[p]T := \text{index}(\text{reduce}(\text{maxindex}, \Delta(x \leftarrow (\alpha, p \land (\text{proc}V(x) = p))) \text{abs}([p, \text{location}(x)] VM) \]
\[ \text{Put}([p]T); \]
\[ \text{Get}(\Delta(x \leftarrow (0:p_{\text{max}})) [x]T); \]
\[ \text{if } p = 0 \text{ then } \text{pivot} := \text{reduce}(\text{max}, \Delta(i \leftarrow P)) [i]T; \]
\[ \text{Put}(\Delta(x \leftarrow (0:p_{\text{max}})) \text{pivot}); \]

For a distributed memory machine this translates to:
\[
[p]T := \text{index}(\text{reduce}(\text{maxindex}, \Delta(x \leftarrow (\alpha, p \land (\text{proc}V(x) = p))) \text{abs}([p, \text{location}(x)] VM) \]
\[ \text{if } p = 0 \text{ then } S := \Delta(i \leftarrow P \setminus \{0\}) \text{Recv}(i, [i]T) \]
\[ \text{else Send}(k, [p]T); \]
\[ \text{if } p = 0 \text{ then } T := \Delta(i \leftarrow P \setminus \{0\}) \text{Recv}(i, [i]T); \]
\[ \text{if } p = 0 \text{ then } \Delta(i \leftarrow P \setminus \{0\}) \text{Send}(i, \text{pivot}) \]
\[ \text{else Recv}(0, \text{pivot}) \]

Optimisations for Content-Dependent Index Sets
The optimisations mentioned above are even more important if these content dependent parameter bindings appear in some iteration in which the content part does not change. For example, consider the following clause:
\[ \Delta(k \leftarrow (0:n)) \cdot \{ \Delta(i \leftarrow (\alpha, R_A)) \diamond \text{Clause}(i) \} \]

This clause can be changed to:
\[ I = (\alpha, R_A); \]
\[ \Delta(k \leftarrow (0:n)) \cdot \{ \Delta(i \leftarrow I) \diamond \text{Clause}(i) \} \]

When computation responsibility is introduced we obtain the following form:
\[ \Delta(p \leftarrow P) \parallel \{ I = (\alpha, R_A(i) \land (\text{proc}(f(i)) = p)); \]
\[ \Delta(k \leftarrow (0:n)) \cdot \{ \Delta(i \leftarrow I) \diamond \text{Clause}(i) \} \]

The required index set \( I \) is only computed once before the parameter binding with \( k \) and therefore the \text{Send/Receive} messages can be determined after \( I \) has
been computed. This optimisation decreases the computation measure $C_M$ considerably, because $I$ is computed only once instead of $n+1$ times. For further details on the implementation of these kind of optimisations the reader is referred to [Koelbel90].

8.6 Annotations and Performance

The translation and optimisation of Booster programs and its annotations is an interplay between compiler and the programmer. The programmer gives information to the compiler by describing annotations. In turn, the compiler returns information in the form of *index overhead, volume, computation, parallel*, and *data transport* measures. In addition, if the programmer does not annotate all shapes or views the compiler can return derived annotations. This can give feedback to the user on interactions and dependencies between various shapes.

8.6.1 Measures and Feedback

The software development process for parallel algorithms in Booster is characterised by the exploration of alternative annotation schemes. For the user of the system it is important to have some absolute or relative measure to compare certain annotations too. An absolute measure can be obtained by executing the generated code on the target architecture. This solution, however, is fairly time-consuming and may not reveal the information needed. It is, for example, interesting to quantify the amount of extra computation that is performed to obtain the correct indices for a given V-cal program. This information is directly related to the functions used in the annotation and their effect on the resulting code.\(^4\) Although this measure does not have a direct relation with the performance of the algorithm on some architecture, it does give the user feedback.

*Index Overhead Measure*

The previously introduced *index overhead measure* is defined relative to some base measure. This base measure is obtained by assuming that for a certain program all membership tests are resolved at compile-time and data-dependent index functions can be computed locally. Albeit that these assumptions are unrealistic, and result in very poor execution programs, it is useful as a reference.

---

\(^4\) For example certain partitionings may result in the distributed evaluation of certain data dependent index functions and consequently induce much more communication-overhead.
Data Transport Overhead Measure

The DTM can be used to compare various annotations statically in order to obtain an indication for total amount of data transport induced by that annotation. Although, we did not elaborate on this the DTM can also be used to distinguish between different types of data transport. The first type of data transport is induced by computation organisation annotations (Use-transport). The second type of data transport is induced by data organisation annotations, for example dynamic view annotations. The third type of data transport results from the evaluation of data-dependent index sets and functions. These different types of data transport can give information on the nature of the transport induced.

It should be noted that the DTM alone is not a good measure for the achievable speed-up or efficiency. If data transport does not lead to link or memory contention a higher DTM may even lead to better performance if it allows for more parallelism. The measure is therefore especially useful to detect potential communication hot-spots instead of an overall measure.

Computation and Parallel Measure

The CM and PM can be used to determine the potential speed-up regardless of the annotations that are used. Once the program is annotated the PM typically becomes larger\(^5\), because potential parallelism is lost when the program is mapped to a machine that has less processors than the optimal number for the program. With CM and PM we have two measures that define an upper and lower bound for the execution time of the program. Therefore measures can also be used to compare different annotations with respect to the amount of parallelism that is lost or introduced.

Volume Measure

The VM describes the amount of memory that is used by the program. This also includes memory that is claimed to implement certain functions that result from annotations. This measure is not especially interesting to the programmer, but can be used by the compiler to avoid excessive use of memory and to make a trade-off between various algorithms. For an example, see Section 7.3.1.

8.6.2 (Semi-)Automatic Annotation Generation

In general the programmer will not specify an annotation for each shape or view that is used in the associated Booster program. In those cases the compiler must decide which annotation to use for these shapes or views. The same

---

\(^5\) Recall that PM in a non-annotated program returns the critical path of the program. In the case of an annotated program the SPMD-program that takes longest is returned.
holds for those data structures that are generated by the compiler to compute certain index sets or functions at run-time. The compiler then decides what annotation to choose. In the following example, we present a possible option for the compiler.

**Example 8.7**

A straightforward solution to this problem is to *conform* the annotations with those that are specified. For example consider:

\[
A[2*i, j] := B[i]*C[j]; \quad \text{B[i]} \leftarrow \text{ProcMem}[i \mod c, i \mod c]
\]

Here the shape \( B \) is annotated, whereas both \( A \) and \( C \), are left indeterminate. As a local optimisation the conformity strategy would annotate (half of the elements of) the first dimension of \( A[2*i, \_] \) in the same way as \( B \):

\[
A[2*i, \_] \leftarrow \text{ProcMem}[i \mod c, i \mod c, \_]
\]

For the annotation of the second dimension of \( A \) and \( C \) no additional information can be derived from the annotation of \( B \). Clauses that precede or succeed this particular clause may supply the needed hints, otherwise a standard annotation can be chosen.

**Feedback**

The programmer receives compiler-generated annotations as feedback for monitoring of the translation and optimisation process. The programmer can use this feedback process to learn more about the relationships between shapes and views in the program. The programmer could start with a simple initial annotation, compile it, and select one or more of the annotations returned by the compiler and add these to the annotations in the next phase. This experimentation process could learn the programmer much about the program and the nature of its parallelism.

**8.7 Examples**

The translation of *Booster* programs with annotations is best explained by considering the *Gaussian Elimination* algorithm with a static- as well as dynamic decomposition. We use the results obtained in the example of Section 7.3.1, which discussed the translation of the algorithm without pivoting and Section 7.3.2, which discussed the translation with pivoting. We consider the following annotation from Section 5.6:

\[
A \leftarrow \text{VM[row_decomposition]}; \quad \text{// with pivoting}
\]

\[
H \leftarrow \text{VM[row_decomposition]}; \quad \text{// without pivoting}
\]
These annotations respectively translate to:

\[ A = \Delta(q \leftarrow I) [\text{block}(q)]A^P \]
\[ H^k = \Delta(q \leftarrow I(k)) [\text{block}(k,q)]H^P \]

Where \( \text{block}(q) = (q_1 \text{ div } c, (q_1 \text{ mod } c), q_2^*c) \). Below, we derive computation responsibility and data management for these annotations.

**Annotation on the shape**

The integration of the annotation is based on the final V-cal program found in Section 7.3.1. By substituting the annotation for \( A \) in the V-cal program, we obtain:

\[
\Delta(i \leftarrow (0:n)) // \{ [[T := i ] \}
\]
\[
\Delta(k \leftarrow (0:n)) \cdot \{
\text{pivot} = \text{reduce}(\text{maxindex}, \Delta(k)) \text{ abs}([[k]T,0] \Delta(q \leftarrow I) [\text{block}(q)]A^P);
[[0]T, [\text{pivot}]T = ([\text{pivot}]T, [0]T);
\Delta(i \leftarrow (0:n-k)) // \Delta(j \leftarrow (0:n-k)) //
\}
[[i+k+1]T,j+k+1] \Delta(q \leftarrow I) [\text{block}(q)]A^P :=
[[i+k+1]T,j+k+1] \Delta(q \leftarrow I) [\text{block}(q)]A^P -
[[i+k+1]T,k] \Delta(q \leftarrow I) [\text{block}(q)]A^P[[k]T,k] \Delta(q \leftarrow I) [\text{block}(q)]A^P^*[[k]T,j+k+1] \Delta(q \leftarrow I) [\text{block}(q)]A^P; \}
\]

Contraction yields:

\[
\Delta(i \leftarrow (0:n)) // \{ [[T := i ] \}
\]
\[
\Delta(k \leftarrow (0:n)) \cdot \{
\text{pivot} = \text{reduce}(\text{maxindex}, \Delta(k)) \text{ abs}(\text{block}[[k]T,0])A^P);
[[0]T, [\text{pivot}]T = ([\text{pivot}]T, [0]T);
\Delta(i \leftarrow (0:n-k)) // \Delta(j \leftarrow (0:n-k)) //
\}
[[\text{block}([i+k+1]T,j+k+1)]A^P := [\text{block}([i+k+1]T,j+k+1)]A^P -
[\text{block}([i+k+1]T,k)]A^P[\text{block}([k]T,k)]A^P^*[[k]T,j+k+1]A^P; \}
\]

The next step is to derive computation responsibility and apply the optimisation discussed in Section 8.5.3 to the computation of the pivot. According to the conformity rule the intermediate data structure \( T \) is annotated in the same way as the first dimension of the shape \( x \):

\[ T = \Delta(\pi_1(q) \leftarrow \pi_1(I)) [\pi_1(\text{block}(q))]A^P \]
This expansion yields the following *V-cal* program:

\[
\delta(p \leftarrow P) \quad \{ \\
\Delta(i \leftarrow (0:n-k \mid \text{block}(i) = p)) \quad \{ [[i]T := i ]; \\
\Delta(k \leftarrow (0:n)) \quad \{ \\
\quad \text{"Compute pivot"} \\
\quad \text{"Swap [0]T and [pivot]T"} \\
\quad \text{"Send Pivot-row"} \\
\Delta(i \leftarrow (0:n-k \mid \text{block}(i) = p)) \quad \Delta(j \leftarrow (0:n-k)) \quad \{ \\
\quad [\text{block}([i+k+1]T,j+k+1)]A^P := [\text{block}([i+k+1]T,j+k+1)]A^P - \\
\quad \text{[block}([i+k+1]T,k)]A^P/[\text{block}([k]T,k)]A^P * \quad \\
\quad [\text{block}([k+k+1]T,j+k+1)]A^P; \} \}
\]

The derivation of the *V-cal* program with *Sends* and *Receives* is comparable to the above derivation.
Chapter 9
Conclusion

The answer to the research question consisted of the Booster language, its annotation language, V-cal, and translation to a parallel program. In this chapter, we will evaluate our approach with the criteria given in Chapter 3 (Section 9.1). Section 9.2 summarises the contributions and limitations of the research reported in this thesis. Section 9.3 discusses the applicability of V-cal to other languages such as Fortran and C. This chapter concludes with a view on future research.

9.1 Evaluation of Booster and Annotations

As the reader is now familiar with the concepts of Booster, we can compare its programming model with various other parallel programming languages and models that were elaborated on in Chapter 2 and the criteria formulated in Chapter 3.

9.1.1 Evaluation Language Criteria

In Chapter 3, it has been shown that Booster was designed starting from a number of criteria. We will have a closer look at whether these criteria are met by Booster and its annotation language.

Information hiding is supported in Booster by the module-construct and the (view-) functions. Compared to conventional languages the programmer can hide details of access to shapes. The second criterion, Abstraction, is supported in Booster by the functional style of programming on the index-domain. In addition, high-level constructs like reduce and the definition of views help to capture recurring patterns and to raise the level of abstraction.

The third criterion, Adding and Preservation of Information, is inherent to the chosen division in Booster language and annotations. The Booster program describes the algorithmic information and the annotation language describes the machine-dependent information. The specification of more annotations augments the machine-dependent information incrementally, which satisfies the fourth criterion: Balance in Gain in Performance and Cost.

The strict separation of machine-dependent constructs in the annotation module and the program in Booster allows for a separation of concerns. The issues of data or computation locality, replication, or incarnation are defined independent of the program. The Booster language only knows a limited number of constructs and concepts: shapes, view- and content-statements and the
encapsulation in (view-)functions and modules. These concepts and constructs are also used in the annotation language to describe the machine-dependent mappings. In this respect Booster adheres to the simplicity criterion.

The annotation language includes two provisions for the scalability of programs. The first provision is the full parameterisation of annotations with respect to the number of processors, memories and sizes of shapes. The second provision is the mapping of the program on the real machine in two stages: first the mapping of the program on the virtual machine, second the mapping of the virtual machine on the real machine. As a consequence, if the number of processors in the real machine increases in the worst case only the mapping between virtual and real machine needs to be changed.

The criteria efficiency, parability, and ease of programming are more difficult to substantiate.

The annotation language provides the programmer with a possibility to influence the translation and thus addresses the efficiency criterion. The true validation of efficiency is the full implementation of a compiler for Booster and the translation of a large number of programs. As this was not possible within the scope of this thesis, we restricted ourselves to the description of the translation and optimisation process for a number of selected examples in Chapters 7 and 8.

The same approach as for the above criterion has been taken for the parability criterion. The examples given in Chapter 5 demonstrate the various mappings from Booster to different types of real machines and illustrate the porting of programs. The support for parability and efficiency does not only consist of a language, but it also needs appropriate tools within a programming environment to allow for program development.

The question whether Booster and its annotation language, or more general view-programming, is suitable for expressing scientific algorithms and eases parallel programming has been validated in the context of the ParTool project. Researchers from several sciences have validated the Booster language for its merits in their particular field from an end-users point of view. Based on their findings changes have been incorporated in the language to include for example annotations on views and sparse shape mappings.

9.1.2 Comparison With Other Approaches

In Chapter 3 a survey and an evaluation for the three categories of languages: explicit, implicit, and enhanced implicit with respect to the criteria was given. Here, we will restrict our discussion to enhanced implicit languages. The approaches are categorised according to the following taxonomy.
9.1 EVALUATION OF BOOSTER WITH CRITERIA

Taxonomy

The annotation language of Booster knows a strict separation between data- and computation-organisation and therefore a range of possible combinations can be defined. Table 9.1 compares the different annotation approaches, where Reside is short for ResideX(p) and Modify is short for ModifyX(p).

Table 9.1 Comparison of different selected approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Type</th>
<th>Coupling</th>
<th>Unstruc Mesh</th>
<th>Dynamic Decompos.</th>
<th>Limitation Decompos.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kali-Fortran</td>
<td>SDSC</td>
<td>—</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Pandore</td>
<td>SDSC</td>
<td>Reside = Modify</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>SUPERB</td>
<td>MDSC</td>
<td>Modify ⊆ Reside</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Dino</td>
<td>MDSC</td>
<td>Reside ⊆ Modify</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Fortran-D</td>
<td>MDSC</td>
<td>Reside = Modify</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Vienna-Fortran</td>
<td>SDSC</td>
<td>Reside = Modify</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>AL</td>
<td>SDSC</td>
<td>Reside = Modify</td>
<td>no</td>
<td>only one dimension</td>
<td></td>
</tr>
<tr>
<td>Booster</td>
<td>—</td>
<td>—</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

The type of annotation SDMC, MDSC, SDMC, and MDMC is defined as follows:

Single Data Single Computation (SDMC)
Here, each data-item of V may be located in multiple memories but data is computed by one processor only: |DataDup(V)| = |V| \land |CompDup(V)| = |V|.

Multiple Data Single Computation (MDSC)
In this case each data-item of V may be located in multiple memories but data is computed by one processor only, or: |CompDup(V)| = |V|.

Single Data Multiple Computation (SDMC)
In this case a each data-item of V may be located in multiple memories but the data-item may be computed by multiple processors, or: |DataDup(V)| = |V|.

Multiple Data Multiple Computation (MDMC)
Multiple memories may contain the same data-element and multiple processors may compute the same value.
The first criterion is the type of annotation (SDSC, SDMC, MDSC, MDMC, or any —). The second criterion is the coupling between Reside and Modify sets, which may vary from no coupling (—) to various restrictions. A third and fourth criterion are the possibility to describe annotations on unstructured meshes and the possibility to describe dynamic annotations. Finally some approaches have built-in limitations.

9.2 Contributions of this Research

The important contribution of this research is perhaps not the Booster language and its annotations, but the identification of a number of issues that are important in (parallel) programming. Section 9.2.1 lists these issues, followed by the implementation status of the Booster compiler in Section 9.2.2. In Section 9.2.3, we elaborate on the limitations of our approach.

9.2.1 Issues in Parallel Programming

The following four issues are raised by the concepts discussed in the thesis:

A machine-independent programming model

The Booster language and annotation language present a machine-independent programming model for a class of algorithms based on array-like data structures. Apart from the provisions for parallel programming, Booster and its annotation language also facilitate the manipulation of data and regions of data. This is an important enhancement, given the increased complexity of data manipulation in scientific algorithms.

The advantages of this approach become more apparent when considering debugging of algorithms. The programmer can observe program behaviour and trace possible errors by using the view-concept from the Booster language. In addition, the programming model separates the issue of "sequential" program correctness from the issue of "parallel" program correctness: once the Booster program is correct, the annotations will not invalidate this.

Separation of concerns

The description of the algorithm is separated from the mapping to the virtual and real machine. This separation allows for incremental parallel program development. The programmer has the option to start with an annotation module containing only a limited number of data- and computation-organisations. If this results — after compilation — in an acceptable target program in terms of speed-up and efficiency, the current annotations suffice. If not, more information on locality of data and computation has to be added to the annotation module. Here, the programmer can make a tradeoff between invested effort and efficiency gained, without having to modify the algorithm itself. The Booster-language and its annotation language thus pair the advantages of ex-
9.2 CONTRIBUTIONS OF THIS RESEARCH

Explicit parallel programming (control over resources), with those of implicit parallel programming (portability and ease of programming).

As compiler-technology improves, the need for more detailed information will disappear and the compiler will generate its own annotations. Moreover, the user can receive high-level feedback from the compiler by inspecting the generated annotations. We will return to this in the next section.

User feedback

Inherent to the choice of programming with annotations is the occurrence of performance bottle-necks on the level of the translated SPMD-program. These performance bottle-necks are again translated in terms of the program and its annotations. Instead of contention in some memory or on some communication links the programmer receives feedback in terms of shapes or views and the associated data and computation organisations. This feedback is given by the Index Overhead, Data Transport, Computation, Parallel, and Volume measures. Each of these measures gives information on performance bottle-necks and directs the programmer to a more improved version of the annotations or even of the algorithm itself.

The compiler-generated annotations are another form of feedback. The programmer will not specify annotations for every shape or view and the compiler can try to induce the most favourable annotations given the restrictions of the program. The incremental definition of annotations and compiler feedback form a method for “parallel” program development within the annotation paradigm.

A formal framework and compilation approach

The Booster language and its annotation part are both based on the unifying formal framework V-cal. The rewrite-rules of V-cal form the basis of the Booster language compiler and possibly other languages and models that translate to V-cal. In Section 9.3, we will elaborate on the more general applicability of V-cal. The advantage of V-cal in comparison with other approaches is that it unifies many aspects of the translation and parallelisation process. In addition, V-cal integrates the representation of the data dependence information through orderings, presenting a single framework for all program transformations.

9.2.2 Implementation Status

A rule-based compiler is used for the implementation of the translation of Booster to Fortran and C. This compiler takes a Booster program and a set of reduction-rules as input and returns a translated version. The advantage of this approach is the direct implementation of the definitions and theorems as
formulated in Chapters 6, 7, and 8 with the reduction-rules. A full implementation is being worked on.

9.2.3 Limitations of the Approach

The programming model presented in this thesis aims at the synergism of portability and ease of programming on the one hand and efficiency on the other. This model, realised by generated communication and synchronisation from data and computation organisation specifications, has its limitations. The major limitation is the inability to express complex synchronisation and communication schemes by the programmer. The compiler translates Booster on a clause-by-clause basis, which yields interleaved communication and synchronisation schemes. Although it is our conviction that these low-level details are part of the compiler-domain, for some algorithms the description of communication and synchronisation in the algorithm itself is necessary. This limitation is not a major deficiency for the domain of scientific algorithms, because according to Fox [Fox88] about 90% of such algorithms can be expressed by annotations.

A second limitation of the language is the static nature of shapes. Dynamic allocation of data is essential for many algorithms and the language needs to contain the corresponding constructs and the compiler needs to implement the corresponding optimisations. Related to this limitation is the so-called meta-view concept. Booster in its current form does not allow for the definition of "shapes of shapes", nor for the definition of abstract data types like stacks, trees, or lists. For most algorithms the lacking functionality can be described with the application modelling formalism Vista [Jong92]. Vista allows for the manipulation of sets of data and can be used to model certain aspects of abstract data types. The question remains however, whether it is desirable to go back to the level of application modelling when encountering low-level dynamic aspects of algorithms.

Despite appearances, this limitation is not fundamental to our concept. The Booster language can be extended with other concepts like object-, set-, parallel- or tuple-space oriented approaches without violating its basic principles. A new formalism on top of Booster can be used to manipulate sets of shapes or views and implement the dynamic nature of some algorithms. The lower-level details can be described in Booster, which will be the subject of further research.

9.3 Generality of the Formal Framework

The formal framework of V-cal is especially suited to describe the semantics and translation of Booster programs. The question that arises is whether V-cal can be used for the translation and optimisation of other languages, as Fortran
9.3 Generality of the Formal Framework

and C. One of the challenging issues is that Booster by its very design does not know hard-to-analyse constructs, such as for-loops and pointers that other languages have.

The use of V-cal and annotations is twofold. The first use is to analyse an explicit parallel program in Fortran and C and try to translate this to a V-cal program with annotations. This is a form of reverse engineering; extract the “meaning” of the specified communication and synchronisation structures. The second use is to analyse an implicit parallel program in Fortran and C and translate this to V-cal. On the resulting V-cal program, one can define annotations to obtain a better parallel version.

The first approach is difficult because in order to find the underlying program and annotations the compiler must analyse explicit communication patterns. The other approach is, especially, for Fortran and (restricted versions of) C more feasible. V-cal is more general than these languages and the translation is rather straightforward. Because these programs have no notion of views and due to language restrictions, essential information is hidden and only limited optimisations are possible.

V-cal Applied to Other Languages

The translation from a given program in an imperative language, such as Fortran or C, to V-cal requires a more intricate strategy. Much information is hidden or even lost in the process of describing an algorithm in one of these languages. We illustrate this point in the following Fortran 77 program fragment, where \( I = (1:n, \text{True}) \):

\[
\text{FOR } i = 1, N \text{ DO} \\
C(i) = B(i-1)*4; \\
A = A + C(i); \\
\text{END}
\]

\[
\Delta(i \leftarrow I) \Diamond \{ [i]C := [i-1]B*4; \\
A := A + [i]C; \}
\]

The straightforward V-cal version of this program illustrates two problems of Fortran. The first problem is the derivation of the partial order \( \Diamond \) for the elements of \( I \). The first assignment in the Fortran-program allows for a parallel ordering, but because of the reduction in the second assignment the ordering is sequential. Consequently the ordering \( \Diamond \) becomes sequential (\( \ast \)) and potential parallelism is lost. This illustrates the restriction the Fortran language can impose on algorithms described in that language. Another option is to split the two assignments in two:

\[
\Delta(i \leftarrow I) \// \{ [i]C := [i-1]B*4 \} \Phi(i > j-1) \\
\Delta(j \leftarrow I) \ast \{ A := A + [j-1]C; \}
\]
Here the first assignment has its parallel ordering and through the inter-clause ordering the potential pipe-lining of the reduction becomes apparent. We can conclude that V-cal offers the definition of “loops” in terms of parameter bindings where the ordering of the index set is user-intended and not imposed by the programming language.

9.4 Future Research

In this thesis the Booster programming language and annotations are applied to a number of examples to demonstrate the feasibility of programming for parallelism with views. Various aspects of Booster and its annotation language, however, require further research. Among these are the applicability and validity of the view-concept in languages such as Fortran and C. Especially Fortran has evolved in the past decade with the definition of Fortran-90 [ANSI89] and PCE-Fortran [PCF90] and is still evolving also in the field of manipulation of large data sets [Bell83, Bell87]. Furthermore, some dialects of Fortran, such as Vectran [Paul75] know the so-called identify-statement that is a restricted version of Booster’s view-statement. Research into a Fortran or perhaps C versions of Booster could facilitate acceptance of the view-concept, while maintaining the advantages of the view-concept.

The translation and optimisation of Booster and its formalisation in V-cal also present a number of open research questions. An important issue is the ordering of transformations in the translation-process. The reduction rules of V-cal include for example vectorization, loop-interchange, and gather-and-scatter operations. The order in which these optimisations are applied determines the efficiency of the resulting optimised program. An option is to evaluate all possible interactions of reduction-rules, but this requires even for small programs exponential time. Therefore, heuristics have to be developed and interacting reductions identified.

The third area of interest for future research is the interaction between programmer and compiler. The programmer describes the Booster program and annotates it. The compiler in turn gives feed-back in the form of measures and annotations on non-specified data structures. This feed-back can be used by the programmer to alter annotations or even the Booster program itself. This feedback process needs further attention and requires a software support environment for experimentation.
Index

Abstraction 19
Algorithm Approach 9
Annotations 11
Arithmetic Functions 54
Arithmetic Operators 52
Assignment 127

Boolean expressions 44
Booleans 33
Bounded Index Sets 105
Bounding Vector 102
Bounding Vectors
  Intersection 104

Card Function 106
Cardinality specification 34
Cartesian Product 102
Choice function 117
Collateral assignment 51
Comparison operators 53
Complex Views 108
Computation Measure 145, 221
Conditional statement 63
Constant definitions 64
Content of a shape 32
Content selection view 43
  abbreviations 45
Content statement 50
  collateral assignment 51
Contraction
  Generalized 123
Control statements 62

Data field 59
Data Organization
  Data dependent mappings 84
  Data duplication 85

Data Structure 106
Dependence relations 57
Difference operator 35
Dim Function 106
Dimension conversions 55
Dimension decreasing view 42
Distributed-memory Machines 3
Domain Propagation Function 106

Ease of programming 6
Efficiency 6
Empty views 50
Expanding view 42

Free variables 39
Function as operator 60
Functional relations
  content statement 55
  dependencies 57

Get Operation 158

Ideal algorithm 9
Index argument 48
Index domain 46
  specifications 41
Index Function 119
Index Overhead Measure 144
Index Propagation Function 106
Index Set
  Cardinality 105, 106
  Dimension 106
  Normalized 105
  Size 106
  Zero-based 105
Index set operators 34
Sparse Structures 85
Index sets
  Bounded 105
  constructors 34
  difference 35
  intersection 35
  range 34
  size function 35
  union 34
Index statement 64
Information Hiding 19
Integer operators 36
Intersect operator 35
Intersection of Bounding Vectors 104
Languages with Extensions 11
Linear programming 71
Load Operation 158
Local shapes 60
Lwb operator 37

Machine Measure 147
Maxindex 54
Measure
  Computation 145, 186, 221
  Data Transport 186
  Index Overhead 185
  Machine 147
  Parallel 146, 186, 222
  Volume 145, 186, 221
Minindex 54
Module template 64
Modules 64
Multi-dimensional operators 53

Non-determinism 52
Non-overlapping Annotations 180
Non-rectangular views 56
Normal Form 125

Operators 52
Ord function 52

Ordering
  Clauses 130
  Parameter Binding 126
Orderings 126
Ordinal Function 103
Orthogonality 20

Parallel Languages 12
Parallel Measure 146, 222
Parallel ordering 159
Parameter 34
Permutation views 41
Position Function 103
Preservation of information 22
Priority for functions 60
Program 9
Projection Function 102
Put Operation 158

Range operator 34
Real machine 74
Receive Operation 158
Recursive Views 111
Reduce 55
Reduction function 55
Reference Function 124
Relation between operations 158
Relational operators 53
Renumber Views 145
Renumbering 39
Repetition statement 62
Reshaping views 42
Rules
  content selection views 46
  free variables 40
  view functions 48
  view identifiers 39

Scalability 21
Selection Functions 54
Selection view 41
INDEX

Send Operation 158
Sequentialization 58
Shape argument 48
Shape constant 59
Shape declarations 34
Shared data-space notion 13
Shared-memory Machines 3
Simple data types 33
Simple View Compositions 37
Simple Views 36, 108
  view functions 50
Simplex method 71
Size function 35, 106
SPMD 153, 161
Statements
  conditional 63
  content 50
  index 64
  repetition 62
  view 38
Store Operation 158
Subroutine Extensions to Languages 12
Superimposed Parallel Models 13

Translation
  Dynamic views 164

Union operator 34
Upb operator 37

View
  Application 109
  Complex 108
  Composition 109
  Inversion 122
  Recursive 111
  Renumber 145
  Simple 108
View Application 109
View functions 47
View identifier 38

View operations 38
View statement 38
  general template 46
  renumbering 39
Views 106
Virtual machine 74
Volume Measure 145, 221
References


Bell83  Bell, J.L., *Flotran user documentation*, National Center for Atmospheric Research, Boulder, USA, July 1983.


REFERENCES


Budd88a Budd, T.A., An APL Compiler, Springer-Verlag, 1988, pp. 156.


Appendix A
Proofs of Theorems

This appendix contains the theorems and their proofs of Section 6.3.2.

**Theorem V6  Parameter Decomposition**
Let $I = J \times K$ and $i = (j,k)$, then

$$
\Delta(i \leftarrow I) [f(i)] ve \iff \Delta(j \leftarrow J) \Delta(k \leftarrow K) [f(j,k)] ve
$$

**Proof:**
We rewrite both parameter bindings and selections as views

$$
\Delta(i \leftarrow I) [f(i)] ve \iff V1 \ \sqrt{(I, id, f)} ve \\
\Delta(j \leftarrow J) \Delta(k \leftarrow K) [f(j,k)] ve \iff V1 \ \sqrt{(J \times K, id, f)} ve = \sqrt{(I, id, f)} ve
$$

Consequently, the parameter bindings may be eliminated.

\square

**Theorem V7  Selection Composition**
Let $Dim(i) = Dim(f)$, then

$$
[f][i] \iff [i]
$$

**Proof:**
We rewrite both selections as views

$$
[f] ve \iff V1 \ \sqrt{(i; j, id, id)} ve \\
[i] ve \iff V1 \ \sqrt{(j; f, id, id)} ve
$$

Composition of the simple views yields a view:

$$
\sqrt{(i; i) \& (j; f, id, id)} \iff \sqrt{(i; i, id, id)} \iff [i].
$$

\square

**Theorem V8  Simplification**
Let $Index(ve) = I$, then

$$
\Delta(i \leftarrow I) [i] ve \iff ve
$$

**Proof:**

$$
\Delta(i \leftarrow I) [i] ve \iff Def \ V1 \ \sqrt{(I, id, id)} ve
$$

The view $\sqrt{(I, id, id)} ve$ with $Index(ve) = I$, defines the identity view, because

$$
\sqrt{((\alpha, P), id, id)} (\alpha, P) = (\alpha \ & id(\alpha), P \circ id \land P) = (\alpha, P)
$$
Consequently, the view may be eliminated and hence the semantically equivalent parameter binding and selection can be eliminated too.

\[ \square \]

\section*{Theorem V9 \hspace{1em} Extraction}

Let \( f \) and \( g \) be functions for which \( \forall_{i,j} (f(i) \subseteq N^b \land g(j) \subseteq N^c) \) then

\[ [f(i),g(j)] \leftrightarrow \Delta(x \leftarrow (b, \{ x = f(i) \})) [x.g(j)] \]

\textbf{Proof:}

According to the denotational convention that a selection may be replaced by a parameter binding with corresponding selection, we obtain the following:

\[ [f(i),g(j)] \leftrightarrow [I \times J, id, (f(g))] \leftrightarrow [I] \Delta(x \leftarrow (b, \{ x = f(i) \})) \Delta(y \leftarrow (c, \{ y = g(j) \})) [x.y] \]

We now apply Theorem VI obtain required version:

\[ \Delta(x \leftarrow (b, \{ x = f(i) \})) \Delta(J, id, (x,g)) \leftrightarrow [I] \Delta(x \leftarrow (b, \{ x = f(i) \})) [x.g(j)] \]

\[ \square \]

The next rewrite rule defines the interaction between selections and parameter bindings and is referred to as a \textit{contraction}, because application of this theorem eliminates one of the selections and the parameter binding involved.

\section*{Theorem V10a \hspace{1em} Contraction for parameter bindings}

If \( Dim(g(i)) = Dim(J), g(i) \in J = (a,P), K = (b,Q), \) and \( g \) and \( f \) are functions, then:

\[ [g(i),\_] \Delta((j,k) \leftarrow J \times K) [f(j,k)] \leftrightarrow \Delta(k \leftarrow K) [f(g(i),k)] \]

\textbf{Proof:}

According to the denotational convention, we may include a parameter binding for \( i \), then:

\[ [g(i),\_] \Delta((j,k) \leftarrow J \times K) [f(j,k)] \leftrightarrow \Delta(l \leftarrow K) [g(i),\_] \Delta((j,k) \leftarrow J \times K) [f(j,k)] \leftrightarrow [I] \Delta(J, id, (g,id)) \Delta(K, id, (f, g(i)), f) \leftrightarrow \text{View Composition} \]

\[ \Delta(K, id, (f(g(i),k))) \leftrightarrow [I] \Delta(k \leftarrow K) [f(g(i),k)] \]

\[ \square \]

\section*{Theorem V10b \hspace{1em} Contraction for parameter bindings (Prove with views)}

If \( Dim(g(i)) = Dim(J), g(i) \in J = (a,P), K = (b,Q(j)), \) and \( g \) and \( f \) are functions, then:
APPENDIX A PROOFS OF THEOREMS

\[ [g(i),.] \Delta((j,k) \leftarrow J \times K(j)) [f(k)] \Leftrightarrow \Delta(k \leftarrow K(g(i))) [f(k)] \]

**Proof:**

According to the denotational convention, we may include a parameter binding for \( i \), then:

\[ [g(i),.] \Delta((j,k) \leftarrow J \times K(j)) [f(k)] \Leftrightarrow \]
\[ \Delta(l \leftarrow K) [g(i),.] \Delta((j,k) \leftarrow J \times K(j)) [f(k)] \Leftrightarrow \text{V1 and V2} \]
\[ \sqrt{(i:i) \times K, id, (g, id)} \sqrt{(J \times K(j), id, f)} \Leftrightarrow \text{View Composition} \]
\[ \sqrt{(K(g(i)), id, f)} \Leftrightarrow \]
\[ \Delta(k \leftarrow K(g(i))) [f(k)] \]

\[ \Box \]

**Interactions with Superscalar functions**

Another important class of interactions is the one between selections and parameter bindings with superscalar functions. This interaction typically results in a change of the dimension of the superscalar function.

**Theorem V11  Selections and Superscalar functions**

Let \( \#^d \) be a \( d \)-dimensional superscalar function and \( \#^e \) its corresponding \( e \)-dimensional function, where \( e = d - \text{Dim}(i) \) then:

\[ [i] \#^d(ve_1, ve_2, ..., ve_n) \Leftrightarrow \#^e([i]ve_1, [i]ve_2, ..., [i]ve_n) \]

**Proof:**

For convenience, let \( K = (b, (i = k)) \).

\[ [i] \#^d(ve_1, ve_2, ..., ve_n) \Leftrightarrow \text{V8} \]
\[ \Delta(k \leftarrow K) [i] \#^d(ve_1, ve_2, ..., ve_n) \Leftrightarrow \text{Def V1} \]
\[ \sqrt{(K, id, id)} \#^d(ve_1, ve_2, ..., ve_n) \Leftrightarrow \text{Def V5} \]
\[ \#^e(\sqrt{(K, id, id)} ve_1, \sqrt{(K, id, id)} ve_2, ..., \sqrt{(K, id, id)} ve_n) \Leftrightarrow \text{Def V5} \]
\[ \#^e([i]ve_1, [i]ve_2, ..., [i]ve_n) \]

\[ \Box \]

**Theorem V12  Parameter Bindings and Functions**

Let \( \#^d \) and \( \#^{d+e} \) be superscalar functions, where \( e = \text{Dim}(K) \) and \( f \) a function:

\[ \Delta(k \leftarrow K) \#^d([f(k)]ve_1, [f(k)]ve_2, ..., [f(k)]ve_n) \Leftrightarrow \]
\[ \#^{d+e}(\Delta(k \leftarrow K) [f(k)]ve_1, \Delta(k \leftarrow K) [f(k)]ve_2, ..., \Delta(k \leftarrow K) [f(k)]ve_n) \]

**Proof:**

The interaction of parameter bindings and superscalar functions is proven as follows:
\[ \Delta(h \leftarrow K) \#^d([f(h)]ve_1, [f(h)]ve_2, \ldots, [f(h)]ve_n) \iff V11 \]
\[ \Delta(h \leftarrow K) [f(h)] \#^c(ve_1, ve_2, \ldots, ve_n) \iff \text{Def V1} \]
\[ \sqrt{(K, id, f)} \#^c(ve_1, ve_2, \ldots, ve_n) \iff \text{Def V5} \]
\[ \#^{d+c}(\sqrt{(K, id, f)} ve_1, \sqrt{(K, id, f)} ve_2, \ldots, \sqrt{(K, id, f)} ve_n) \iff \text{Def V1} \]
\[ \#^{d+c}(\Delta(h \leftarrow K) [f(h)]ve_1, \Delta(h \leftarrow K) [f(h)]ve_2, \ldots, \Delta(h \leftarrow K) [f(h)]ve_n) \]

\[ \Box \]

**Interactions with reduces**

The next class of interactions is those between views, selections, and parameter bindings on one hand and the reduce construct on the other hand.

**Theorem V13 View and Reduce**

Let \( \#^e \) and \( \#^d \) be superscalar functions, where \( e = \text{Dim}(ip) \) and \( d = \text{Dim}(I) + e \), then

\[ \sqrt{(I, dp, ip)} \text{reduce}(\#^e, \Delta(h \leftarrow K)) ve(k) \iff \]
\[ \text{reduce}(\#^d, \Delta(h \leftarrow K)) \sqrt{(I, dp, ip)} ve(k) \]

**Proof:**

The equation is proven by expanding the reduce function:

\[ \sqrt{(I, dp, ip)} \text{reduce}(\#^d, \Delta(h \leftarrow K)) ve(k) \iff \text{V3} \]
\[ \sqrt{(I, dp, ip)} ((\text{reduce}(\#^d, \Delta(h \leftarrow U)) ve(k)) \#^d (\text{reduce}(\#^d, \Delta(h \leftarrow V)) ve(k))) \]

Application of Definition V5 yields:

\[ ((\sqrt{(I, dp, ip)} \text{reduce}(\#^d, \Delta(h \leftarrow U)) ve(k)) \#^e \]
\[ ((\sqrt{(K, dp, ip)} \text{reduce}(\#^d, \Delta(h \leftarrow V)) ve(k))) \]

By induction, we get:

\[ \text{reduce}(\#^e, \Delta(h \leftarrow K)) \sqrt{(I, dp, ip)} ve(k) \]

\[ \Box \]

Selections and reduces effectively decreasing the dimension of the reduce-function \( \# \) which is close to intuition.

**Theorem V14 Selection and Reduce:**

Let \( \#^d \) be a 2-ary superscalar function which takes two arguments with equally shaped index sets of dimension \( d \), and \( \#^e \) its equivalent of dimension \( e = d - \text{Dim}(f) \) then:

\[ [j] \text{reduce}(\#^d, \Delta(h \leftarrow K)) ve(k) \iff \text{reduce}(\#^e, \Delta(h \leftarrow K)) [j] ve(k) \]

**Proof:**

This equation is proven as follows:
[\forall] \text{reduce}(\#^d, \Delta(k \leftarrow K)) \text{ve}(i,k) \iff \forall^3

[\forall] (\text{reduce}(\#^d, \Delta(k \leftarrow L)) \text{ve}(i,k)) \#^d (\text{reduce}(\#^d, \Delta(k \leftarrow J)) \text{ve}(i,k))

Application of Theorem V11 yields the following:

\(([\forall] \text{reduce}(\#^d, \Delta(k \leftarrow L)) \text{ve}(i,k)) \#^e ([\forall] \text{reduce}(\#^d, \Delta(k \leftarrow J)) \text{ve}(i,k))\)

By repeating the application of Theorem 12 we obtain, after a finite number of steps a form where the size of the sets L and J is equal to one. According to Definition 5 the reduce is then equal to the view expression. This implies that:

\(\text{reduce}(\#^e, \Delta(k \leftarrow \{c:c\})) ([\forall] \text{ve}(i,k)) = ([\forall] \text{ve}(i,c))\)

In the next step, we use Definition V5 in reverse:

\(\text{reduce}(\#^e, \Delta(k \leftarrow \{c:c\})) ([\forall] \text{ve}(i,k))\)

Theorem 12 is applied in reverse yielding eventually:

\(\text{reduce}(\#^e, \Delta(k \leftarrow K)) ([\forall] \text{ve}(i,k))\)

\(\Box\)

Theorem V15  Parameter Binding and Reduce

Let \#^d, and \#^e be superscalar functions, where \(e = \text{Dim}(I) + d:\)

\(\Delta(i \leftarrow I) \text{reduce}(\#^d, \Delta(k \leftarrow K)) \text{ve}(i,k) \iff \text{reduce}(\#^e, \Delta(k \leftarrow K)) \Delta(i \leftarrow I) \text{ve}(i,k)\)

Proof:

This equation is proven as follows, where \(L \cap J = \emptyset,\) and \(L \cup J = K\)

\(\Delta(i \leftarrow I) \text{reduce}(\#^d, \Delta(k \leftarrow K)) \text{ve}(i,k) \implies \text{Def V4}\)

\(\Delta(i \leftarrow I) (\text{reduce}(\#^d, \Delta(k \leftarrow L)) \text{ve}(i,k)) \#^d (\text{reduce}(\#^d, \Delta(k \leftarrow J)) \text{ve}(i,k))\)

Application of Theorem V12 yields the following:

\((\Delta(i \leftarrow I) \text{reduce}(\#^d, \Delta(k \leftarrow L)) \text{ve}(i,k)) \#^e\)

\((\Delta(i \leftarrow I) \text{reduce}(\#^d, \Delta(k \leftarrow J)) \text{ve}(i,k))\)

By repeating the application of Theorem 12 we obtain, after a finite number of steps a form where the size of the sets L and J is equal to one. According to Definition V5 the reduce is then equal to the view expression. This implies that:

\(\Delta(i \leftarrow I) \text{reduce}(\#^d, \Delta(k \leftarrow \{c:c\})) \text{ve}(i,k) = \Delta(i \leftarrow I) \text{ve}(i,c)\)

In the next step, we use Definition V5 in reverse:

\(\text{reduce}(\#^e, \Delta(k \leftarrow \{c:c\})) \Delta(i \leftarrow I) \text{ve}(i,k)\)
Theorem 12 is applied in reverse yielding eventually:

\[ \text{reduce}(\#, \Delta(h \leftarrow K)) \Delta(i \leftarrow I) \vee i, k) \]

\[ \square \]

**Theorem V16  Interchange Reduce and Parameter bindings**

Let \( \# \) be a superscalar function and \( |(a, P(i, k))| \leq 1 \), then

\[ \text{reduce}(\#, \Delta(h \leftarrow K)) \Delta(i \leftarrow (a, P(i, k))) \vee i, k) \leftrightarrow \text{reduce}(\#, \Delta(i \leftarrow (a, \text{True})) \text{reduce}(\#, \Delta(h \leftarrow (K, P(i, k)))) \vee i, k) \]

**Proof:**

\[ \text{reduce}(\#, \Delta(h \leftarrow K)) \Delta(i \leftarrow (a, P(h))) \vee i, k) \leftrightarrow \forall x \]

\[ T(0) \# T(1) \# T(2) \# \ldots \# T(\text{Size}(K)-1) \quad (\text{where } T(h) = \Delta(i \leftarrow (a, P(i, k))) \vee i, k)) \]

Because \( \# \) is associative and commutative we can interchange the terms \( T(x) \) and split them in two groups where \( E(h) = T_1(h) \# T_2(h) \# \ldots \# T(\text{Size}(K)) \), where

\[ P(i, k) = \text{False} \]

\[ F(h) = T_{n+1}(h) \# T_2(h) \# \ldots \# T_{n+m}(h) \], where \( P(i, k) = \text{True} \)

The \( E(h) \) can be eliminated because \( \emptyset \# \emptyset = \emptyset \). \( F \) can be regrouped such that all \( h \) for which \( P(i, k) = \text{True} \) are grouped. Because for each \( i \) there may be more than one \( h \), we formulate this in terms of a \text{reduce}:

\[ \text{Let } S(i) = \text{reduce}(\#, \Delta(h \leftarrow (K, P(i, k))) \vee i, k) \]

\[ S(0) \# S(1) \# S(2) \# \ldots \# S(\text{Size}(I)-1) \]

The last step is fairly straightforward:

\[ \text{reduce}(\#, \Delta(i \leftarrow I)) \text{reduce}(\#, \Delta(h \leftarrow (K, P(i, k)))) \vee i, k) \]

\[ \square \]

**The inverse of views**

A view \( V = \sqrt[\perp]{K, dp, ip} \) \( A \) defines an onto and one-to-one relation from the index set of \( A \) to the index set of \( V \). From the index set of \( V \) the view \( \sqrt[\perp]{} \) defines a total function to the index set of \( A \). The "inverse" of \( \sqrt[\perp]{} \): \( A = \sqrt[inv]{K, dp, ip} \) \( V \) defines a relation from the index set of \( V \) to \( A \) and can therefore not be described as \( A = \sqrt[inv]{K, dp, ip} \) \( V \), because single element of \( A \) may be related to multiple elements of \( V \). We can, however, give a view expression that yields all elements of \( V \) that are associated with an element of \( A \).
Theorem V17 "Inverse" of Views

Let $V = \sqrt{((b,R),id,ip) A}$ be a view application and $I = Index(A)$, then each element of $A$, $[i]A$ is associated with:

$$\Delta(t \leftarrow (b,R \land ip(t) = i)) [t] V$$

Proof:

According to the definition of a parameter binding and selection we may write:

$$V = \sqrt{((b,R),id,ip) A} \iff \Delta(j \leftarrow J) [ip(j)] A$$

By substituting $V = \sqrt{((b,R),id,ip) A}$ in the expression $[i]A = \Delta(t \leftarrow (b,R \land ip(t) = i)) [t] V$, we should obtain an equality:

$$[i] A = \Delta(t \leftarrow (b,R \land ip(t) = i)) [t] V \iff Substitution$$

$$\Delta(t \leftarrow (b,R \land ip(t) = i)) [t] \Delta(j \leftarrow J) [ip(j)] A \iff Contraction$$

$$\Delta(t \leftarrow (b,R \land ip(t) = i)) [ip(t)] A$$

According to Theorem V9 (Extraction) this may be reduced to:

$$[i] A = [i] A$$

This concludes the proof.

$\Box$
Appendix B
Measures

Volume Measure

An important notion in parallel programming is the amount of memory that is needed for the computation. Especially, for programs that have large data structures it is essential that they fit into (local) memory. We therefore introduce the Volume Measure (VM) that returns the total memory requirement for a V-cal program.

The following union operator ∪ unifies sets of tuples ⊆ pow(Var × N), where Var ⊆ Σ*. The operator is defined as follows:

Definition
Let ∪ : pow(Var × N) × pow(Var × N) → pow(Var × N) and U, W ∈ pow(Var × N), then:

\[ U ∪ W = \{ (v, s) \mid (v, s) ∈ U ∨ (v, s) ∈ W \} \]

Definition
Let VM : pow(V-cal-programs) → N and SIZES : pow(V-cal-programs) → (Var × N), then

SIZES(DataStructure) = (DataStructure.Id, size(DataStructure.Index))
SIZES(⟨ (K, dp, ip) ve ⟩) = SIZES(ve)
SIZES(f(ve₁, ve₂, ..., veₙ)) = ∪ᵢ SIZES(veᵢ)
SIZES(Stat) = VM(Stat.Left) ∪ SIZES(Stat.Right)
SIZES(Clause₁ ∘ Clause₂) = SIZES(Clause₁) ∪ SIZES(Clause₂)

VM( Program ) = ∑_{ (var, s) ∈ SIZES(Program) } s

Computation Measure
The computation measure returns the number of operations in a program. This measure is subdivided in a measure for computations in the index domain and in the data domain. This measure does not give a very precise estimate of the actual computation, but gives an indication of the complexity. It is very useful for the comparison of two versions of the same program, because it can give a relative measure.
Definition
Let \( CM, CM_I, CM_D : pow(V-cal-programs) \to N \)

The computation measure \( CM_D \) for the data domain is defined as:

\[
CM_D(\text{reduce}(\#, \Delta(k \leftarrow K)) ve(k)) = \Sigma_{k \in K} CM_D(ve(k)) + (|K| - 1) * CM_D(#) \\
CM_D(f(ve_1, ve_2, ..., ve_n)) = \Sigma CM_D(ve_i) + CM_D(f) \\
CM_D(\text{Stat}) = CM_D(\text{Stat.Right}) \\
CM_D(\text{Clause}_1 \Phi \text{Clause}_2) = CM_D(\text{Clause}_1) + CM_D(\text{Clause}_2)
\]

The computation measure \( CM_I \) for the index domain is defined as:

\[
CM_I(\Delta(k \leftarrow (b,R)) ve(k)) = \Sigma_{k \in K} CM_I(ve(k)) + |N^b| * CM_I(R) \\
CM_I([f(i)] ve) = CM_I(f) + CM_I(ve) \\
CM_I(\text{reduce}(\#, \Delta(k \leftarrow K)) ve(k)) = \Sigma_{k \in K} CM_I(ve(k)) \\
CM_I(f(ve_1, ve_2, ..., ve_n)) = \Sigma_i CM_I(ve_i) \\
CM_I(\text{Stat}) = CM_I(\text{Stat.Left}) + CM_I(\text{Stat.Right}) \\
CM_I(\text{Clause}_1 \Phi \text{Clause}_2) = CM_I(\text{Clause}_1) + CM_I(\text{Clause}_2)
\]

The computation measure \( CM \) is the sum of the two measures:

\[ CM(\text{program}) = CM_I(\text{program}) + CM_D(\text{program}) \]

Parallel Measure
The ultimate goal for a Booster program is to execute as many operations in parallel as possible. In the optimisation process of V-cal programs we therefore need a measure for the degree of parallelism in the program. This measure is referred to as the Parallel Measure (PM) and is defined as follows:

Definition
Let \( PM : pow(V-cal-programs) \to N \)

\[
PM(\text{DataStructure}) = 1 \\
PM(f(ve_1, ve_2, ..., ve_n)) = \max_i (PM(ve_i)) \\
PM(\text{reduce}(\#, \Delta(k \leftarrow K)) ve(k)) = \log_2(\text{size}(K)) * CM(#) + \max_i (PM(ve_i)) \\
PM(\text{Stat}) = PM(\Delta(i \leftarrow I) \circ \text{Stat}(i)) = \\
\Sigma_j \max(k \in K(j) \mid PM(\text{Stat}(k))) \\
PM(\text{IfStat}) = PM(\text{Cond}) + \max(PM(\text{Then}), PM(\text{Else})) \\
PM(\text{WhileStat}) = PM(\text{Cond}) + PM(\text{Body}) \\
PM(\text{Clause}_1 \Phi \text{Clause}_2) = PM(\Delta(i \leftarrow I) \bullet (\text{Clause}_1(i) || \text{Clause}_2(i))) = \\
PM(\Sigma_i \max(PM(\text{Clause}_1(i)), PM(\text{Clause}_2(i))))
\]
Appendix C
Booster Syntax

Conventions

- Booster reserved words and literal tokens are in **boldface**;
- Non-terminals are in *italics*;
- Meta-symbols are in normal typeface;

**Booster reserved words**

*Language keywords*

all, and, begin, cart, div, do, else, end, function, if, int, intersect, iter, lwb, mod, module, not, or, real, reduce, shape, then, to, type, union, upb, view, while

*Standard functions*

sin, cos, tan, sqrt, ln, abs, max, min, maxindex, minindex

*Special symbols*

+ - * / < <= = <> >= > ( ) [ ] {} # : @ ! & ~ ^ $ _ := <= -> , ; .

**The syntax**

**SEMI-TERMINALS**

- letter := a | b | c | ... | z | A | B | C | ... | Z
- digit := 0 | 1 | 2 | ... | 9
- identifier := letter { letter | digit | _ } 
- value := intValue | realValue
- realValue := intValue [ . digit { digit } ] [ E | e ] intValue
- intValue := [ - ] digit ( digit )
- standardFunction := sin | cos | tan | sqrt | ln | abs | max | min

- binaryOp := + | - | * | /
- binaryIntOp := + | - | * | div | mod
- unaryOp := + | -
- unaryIntOp := + | -
relationOp ::= = | <> | < | <= | > | >=
indexOp ::= # | cart
binaryRangeOp ::= : | to
offsetOp ::= @
binarySetOp ::= | | union | & | intersect
unarySetOp ::= ~ | not
allOp ::= _ | all
selectionOp ::= ^ | lwb | $ | upb
binaryBoolOp ::= and | or
unaryBoolOp ::= not

All non-terminals ending in Id are identifiers:

somethingId ::= identifier

All non-terminals ending in List are either comma- or semicolon-separated lists, but never both. There is no formal way of describing this; the closest we can get is:

somethingList ::= something | somethingList delimiter something
delimiter ::= , | ;

with the additional requirement that each list chooses either comma or semicolon as a delimiter - they can not be mixed in one list, nor be chosen different for different occurrences of somethingList.

The rule for determining the type of delimiter used by lists encountered in the syntax: if the items enumerated in the list are surrounded by literal brackets ([ ] or ( )) then the list must be comma-separated. If no such brackets are specified then the list is semi-colon separated.

MODULE DEFINITION

module ::= module moduleId [ ( argIdList ) ] -> ( argIdList ) ;
shapeDeclList ;
[ const constDeclList ; ]
[ viewFunctionDeclList ; ]
[ functionDeclList ; ]
[ shape shapeDeclList ; ]
begin
[statementList ; ]
end.

SHAPE DECLARATION

shapeDecl ::= shapeIdList ( indexSet ) of typeSpecifier

indexSet ::= indexSet cartOp integerExpr |
integerExpr

integerExpr ::= integerExpr binaryIntOp integerExpr |
unaryIntOp integerExpr |
sqrt ( integerExpr ) |
( integerExpr ) |
intValue |
parameterId

typeSpecifier ::= int | real

VIEW FUNCTION DEFINITION

viewFunctionDecl ::= view function viewFunctionId ( argIdList ) ->
( argIdList ) ;
viewArgDeclList ;
begin

[ viewStatementList ; ]
end

viewArgDecl ::= viewIdList ( indexSet )

FUNCTION DEFINITION

functionDecl ::= function functionId ( argIdList ) -> (argIdList ) ;
shapeDeclList ;
[ shape shapeDeclList ; ]
begin

[ statementList ; ]
end

STATEMENT DEFINITION

statement ::= viewStatement |
contentStatement |
controlStatement
VIEW STATEMENT DEFINITION

viewStatement ::= ( viewIdList ) <- viewFunctionCall |
                 viewLValue <- viewExpr

viewLValue ::= viewId [ ( indexSet ) ] viewApplication ]

viewExpr ::= viewRTerm |
           viewRTerm viewApplicationList |
           viewRTerm ( indexSet ) viewApplication |
           viewRTerm viewApplicationList ( indexSet ) viewApplication

viewRTerm ::= viewId | shapeId | viewFunctionCall

viewFunctionCall ::= viewFunctionId ( viewExpr )

viewApplication ::= [ indexSelectionList ]

indexSelection ::= ( indexSelection ) |
                 unarySetOp indexSelection |
                 indexSelection binarySetOp indexSelection |
                 polyIndex

polyIndex ::= ( polyIndex ) |
             dataIndexFunction |
             singleIndex binaryRangeOp singleIndex |
             singleIndex offsetOp integerExpr |
             allOp |
             singleIndex

singleIndex ::= singleIndex binaryIntOp singleIndex |
              ( singleIndex ) |
              selectionOp |
              parameterId |
              freeVarId |
              intValue

dataIndexFunction ::= relationOp contentExpr

CONTENT STATEMENT DEFINITION

contentStatement ::= shapeId [ viewApplicationList ] := contentExpr |
                   viewId [ viewApplicationList ] := contentExpr |
                   viewFunctionCall [ viewApplicationList ] :=
                   contentExpr
contentExpr ::= contentExpr binaryOp contentExpr | unaryOp contentExpr | ( contentExpr ) | standardFunction ( contentExpr ) | reduce ( binaryOp, contentExpr ) | viewExpr | multiDimValue

multiDimValue ::= valueList | { multiDimValueList }

CONDITIONAL EXPRESSIONS

condExpr ::= condTerm relOp condTerm | unaryBoolOp condExpr | condExpr binaryBoolOp condExpr | ( condExpr )

condTerm ::= contentExpr | size ( contentExpr )

CONTROL STATEMENT DEFINITION

controlStatement ::= loopStatement | ifStatement

loopStatement ::= iter integerExpr do
               [ statementList ; ]
               end | while condExpr do
               [ statementList ; ]
               end | iter integerExpr while condExpr do
               [ statementList ; ]
               end

ifStatement ::= if condExpr then
               [ statementList ; ]
               [ else
                  [ statementList ; ] ]
               end
Samenvatting
(Parallel Programmeren en Efficiënt Vertalen)

Het in dit proefschrift gepresenteerde onderzoek is uitgevoerd bij TNO te Delft in het kader van onder andere het ParTool project en diverse Esprit projecten (HiParTool en Genesis). Het doel van het ParTool project is het ontwikkelen van een programmeer-omgeving voor parallele computers. Binnen het project zijn er drie subprojecten die tools en methoden voor deze omgeving ontwikkelen: applicatie modellering, algoritme beschrijving en architectuur modellering. Het hier beschreven onderzoek vond plaats op het gebied van algoritme beschrijvingen.

Het programmeren van parallele machines is gecompliceerder dan het programmeren van sequentiële machines. De parallele machines verschillen in aantallen processoren, geheugens en vooral in de verbindingen tussen deze elementen. Voor effectief gebruik van deze machines moet men bij het programmeren zo optimaal mogelijk gebruik maken van de eigenschappen van zo'n parallele machine.

Gebruikers van parallele machines zijn echter niet geïnteresseerd in de opbouw van deze machines, maar willen hun software zo goed en goedkoop mogelijk executeren. Daarbij is het van groot belang dat de software met relatief weinig moeite van het ene type naar het andere type machine kan worden overgezet, vooral omdat deze verschillende typen machines elkaar nogal snel opvolgen.

De twee tegenstrijdige eisen om zo efficiënt mogelijk gebruik van de machine te maken en tevens de software makkelijk over te kunnen zetten, zijn in dit proefschrift in een taal Booster en de bijbehorende "annotatie" taal samengebracht. De programmeur beschrijft het algoritme in de Booster taal, waarna in de annotatie-taal informatie over de beste verdeling van het werk over de verschillende processoren en data over de geheugens kan worden toegevoegd. Deze annotaties geven de vertaler aanwijzingen om efficiëntere code te verkrijgen.

De Booster taal richt zich in het bijzonder op het beschrijven van algoritmen die werken op geindexeerde data structuren (ook wel shapes genoemd). Aangezien de meeste algoritmen voor bewerkingen op deze shapes meestal slechts (regelmatige) delen van de shapes nodig hebben kan men in Booster
zogenaamde *views* definieren. Met een view kan men relaties zoals selecties, 
vervormingen, permutaties, etc. definieren op data structuren.

Het view-principe wordt eveneens gebruikt voor het beschrijven van de 
afbeeldingen van op de (parallele) machines in de annotatie-taal. De shape die 
in het algoritme als data structuur wordt gebruikt is een view op de geheugens 
en processoren van de onderliggende machine. Door het definiëren van 
verschillende relaties tussen deze shapes en de machine kan de programmeur 
de vertaling beïnvloeden. Op deze manier kan het *Booster* programma 
eenvoudig van de ene naar de andere machine worden overgezet en kan de 
efficiëntie van het resulterende parallele programma worden gegarandeerd.

Om de integratie en vertaling van *Booster* en de annotaties mogelijk te 
maken is in dit proefschrift een mathematisch formalisme (*V-cal*) 
gepresenteerd, die de semantiek van *Booster* en de annotatie-taal beschrijft. 
Deze calculus kent zoals *Booster* views en binnen deze calculus zijn een aantal 
eigenschappen van views afgeleid, zo levert bijvoorbeeld de compositie van twee 
views weer een view op. Verder kent deze calculus een aantal uitbreidingen die 
het efficiënt vertalen naar bepaalde machines ondersteunen.

Om de efficiëntie van een bepaalde constructie in *Booster* in combinatie met 
zekere annotaties te “meten”, kent de calculus een aantal indicatoren. Deze 
geven de programmeur en de vertaler informatie over de gevolgen op de 
efficiëntie van een bepaalde constructie in *V-cal*. De indicatoren geven 
bijvoorbeeld aan wat het totale beslag op het geheugen is, wat het maximale 
parallelisme is, en wat de omvang van het data transport is. De indicatoren 
quantificeren op deze manier de aard van inefficiënties aan de hand van het 
*Booster* programma en de annotaties. Het voordeel is dat dit op het denk- 
en werk-niveau van de programmeur gebeurt.

De annotatie-benadering van parallel programmeren heeft drie belangrijke 
voordelen. Het eerste voordeel is het feit dat het annoteren van programmaas 
dichter bij de intuïtie van de programmeur ligt dan het expliciet parallel 
programmeren. Het tweede voordeel is behoud van correctheid, want een 
correct *Booster* programma levert altijd een correct parallel programma op. 
Annotaties beïnvloeden alleen de efficiëntie, maar niet de correctheid. Dit is een 
groot voordeel gezien het feit dat het debuggen van een parallel programma een 
moeilijk probleem is. Het derde voordeel is dat de vertaler informatie aan de 
gebruiker over de uitgevoerde optimalisaties in de vorm van annotaties kan 
teruggeven. Dit kan tot een beter begrip van het gedrag van het beschreven 
algemene leiden.
Curriculum Vitae
