Semantics for Compiling Data Parallelism

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft,
on gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,
on dinsdag 3 maart 1998 te 13:30 uur

door

Paulus Fernandus Gerardus DECHERING

doctorandus in de informatica
geboren te ’s-Hertogenbosch
Dit proefschrift is goedgekeurd door de promotoren:
Prof.dr.ir. H.J. Sips
Prof.dr. J.-J.Ch. Meyer

Samenstelling promotiecommissie:

Rector Magnificus (voorzitter)       Technische Universiteit Delft
Prof.dr.ir. H.J. Sips (promotor)    Technische Universiteit Delft
Prof.dr. J.-J.Ch. Meyer (promotor)  Universiteit van Utrecht
Prof.dr.ir. J. van Katwijk          Technische Universiteit Delft
Prof.dr. A. de Bruin                Erasmus Universiteit Rotterdam
Prof.dr.ir. M.J. Plasmeijer         Katholieke Universiteit Nijmegen
Prof.drs. M. Boasson                Universiteit van Amsterdam /
Dr. E.M. Paalvast                    Hollandse Signaalapparaten BV
                                      Bakkenist Leiderdorp

Printed by:
Universal Press
De Richtoek 36
3901 KS Veenendaal
The Netherlands

Cover design: Huib van Nimmerdor

ISBN 90-9011361-4
NUGI 855

Copyright © 1998 by P.F.G. Dechering

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

The author was affiliated to NWO and was sponsored by SION.

NWO
To my grandfather

who introduced me to science
Contents

Acknowledgements v

Summary vii

1 Introduction 1
  1.1 Semantics and compilers ......................................... 1
  1.2 Research question ................................................. 2
  1.3 Data parallelism versus other parallelism ..................... 2
  1.4 Using data ....................................................... 4
  1.5 Towards a compiler framework .................................. 6
  1.6 Intermediate formalisms ....................................... 8
  1.7 Semantics ....................................................... 10
  1.8 Research question revisited .................................. 12
  1.9 Thesis overview ............................................... 13

2 V-nus and its informal semantics 15
  2.1 The idea behind V-nus ......................................... 15
  2.2 Definitions .................................................... 17
    2.2.1 Notational conventions .................................. 17
    2.2.2 Representation ........................................... 18
    2.2.3 Symbol table ............................................. 18
  2.3 Program constructs ........................................... 18
    2.3.1 Units ..................................................... 18
    2.3.2 Declarations ............................................ 19
    2.3.3 Flow of control ....................................... 20
    2.3.4 Communication ......................................... 20
    2.3.5 Statements ............................................... 22
    2.3.6 Expressions ............................................. 24
  2.4 Defining the semantics ....................................... 26

3 The semantic states 27
  3.1 Introduction ................................................... 27
    3.1.1 Notational conventions .................................. 29
3.2 Towards a program state ........................................... 29
3.3 The construction of the program state ......................... 32
  3.3.1 Scopes and locations ....................................... 32
  3.3.2 The view state ............................................. 33
  3.3.3 The program state .......................................... 34
3.4 Using indexes ..................................................... 36
3.5 The declaration state ........................................... 37
3.6 Access and modification functions ............................ 38

4 The denotational semantics ........................................ 43
  4.1 Introduction ................................................... 43
    4.1.1 Notational conventions ................................. 45
  4.2 The initial states ............................................. 46
  4.3 Expressions ..................................................... 48
    4.3.1 Selections .................................................. 52
  4.4 Views .......................................................... 54
    4.4.1 The view expression .................................... 54
    4.4.2 The view statement ..................................... 58
    4.4.3 The assignment ........................................... 58
  4.5 Functions and procedures .................................... 59
    4.5.1 Parameter passing ....................................... 59
    4.5.2 The function call ....................................... 62
    4.5.3 The procedure call ..................................... 64
  4.6 Selections revisited .......................................... 64
  4.7 Flow of control ............................................... 65
    4.7.1 Sequential composition .................................. 65
    4.7.2 Conditional flow ........................................ 66
    4.7.3 Iterated flow ............................................ 67

5 Implicit parallelism ................................................ 73
  5.1 Introduction ................................................... 73
    5.1.1 Notational conventions .................................. 74
    5.1.2 The forall statements .................................. 74
    5.1.3 Chapter overview ....................................... 75
  5.2 Different types of iteration ................................ 75
  5.3 Related work .................................................. 76
  5.4 The semantics of forall ...................................... 79
    5.4.1 Relating program states ................................ 82
  5.5 The implementation of forall ................................ 84
    5.5.1 Pragmas .................................................. 85
    5.5.2 Annotating the forall term ............................. 85
    5.5.3 The implementation compared to the semantics ...... 86
  5.6 The forall compared to other languages ...................... 88
CONTENTS

5.7 The semantics of Foreach .................................................. 91
5.8 Other implicit parallel constructs ........................................... 93
5.9 Conclusion ................................................................. 95

6 Explicit parallelism .......................................................... 97
6.1 Introduction ............................................................... 97
6.2 The process state .......................................................... 99
6.3 The meaning of a process ................................................... 101
6.4 Semantics of communications .............................................. 103

7 A calculus of transformations ............................................... 107
7.1 Introduction ............................................................... 107
7.1.1 Notational conventions ............................................... 108
7.2 The calculus V-cal ......................................................... 108
7.3 Effects of using V-cal ..................................................... 109
7.4 The specification of transformation rules ................................ 110
7.5 Applying transformation rules ............................................ 114
7.6 A case study ............................................................... 117
7.7 Discussion ................................................................. 120

8 Discussion ................................................................. 123
8.1 Conclusions ............................................................... 123
8.2 Future work ............................................................... 124

A The V-nus grammar ......................................................... 127
A.1 Units ................................................................. 128
A.2 Declarations ............................................................ 128
A.3 Flow of Control .......................................................... 129
A.4 Communication .......................................................... 129
A.5 Statements .............................................................. 130
A.6 Expressions ............................................................. 130
A.7 Symbol Table ............................................................ 131
A.8 Miscellaneous ........................................................... 133
A.9 Tokens ................................................................. 133
A.10 White Space ............................................................ 134

B Function definitions ......................................................... 135
B.1 The meaning function $M_{\Sigma}$ ........................................ 135
B.2 The meaning function $M_{\Gamma}$ ........................................ 136
B.3 The meaning function $M_{\Psi}$ .......................................... 137
B.4 The semantic function $\mathcal{E}$ .......................................... 137
B.5 The semantic function $\mathcal{Loc}$ ....................................... 138
B.6 The semantic function $\mathcal{IP}$ ........................................ 138
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C  Examples of <em>forall</em> statements</td>
<td>141</td>
</tr>
<tr>
<td>D  Transformation rule verification</td>
<td>147</td>
</tr>
<tr>
<td>D.1 Construct substitution</td>
<td>147</td>
</tr>
<tr>
<td>D.2 Communication Insertion</td>
<td>148</td>
</tr>
<tr>
<td>E  Table of symbols</td>
<td>151</td>
</tr>
<tr>
<td>Bibliography</td>
<td>153</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>161</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>163</td>
</tr>
<tr>
<td>Index</td>
<td>164</td>
</tr>
</tbody>
</table>
Acknowledgements

Semantics, in the most general context, is a lovely doctrine to explore. Aggravated on programming languages makes it an attractive research vehicle in the area of computer science. When this subject is combined with compiler construction, one can investigate and exploit an interesting interaction of two fields of research.

First of all, I want to thank my advisor Henk Sips who created the above described context. I am deeply grateful for having had the complete freedom in choosing the directions I wanted to explore. He has also given lots of opportunities for me to present my work in international communities. The atmosphere and possibilities he created made the past four years a truly enjoyable time.

Special thanks go to my second advisor John-Jules Meyer. He introduced me to the field of semantics before I started my PhD. research. The way he taught me semantics, made this subject an attractive field of research for me. I was happy to hear that he wanted to be involved in this thesis. I really appreciate the effort and time he has taken to discuss my ideas on semantics.

When I just started at the Delft University, Leo Breebaart introduced me to work I would be going to use as a starting-point. I want to thank him for that, but besides that, I value his advise: “write down all your ideas and distribute them among the group members”. Doing so really gave me a speed-up in my research. Working with him was great as he was always willing to help me with all kinds of questions.

In the first two years of my research I had the pleasure to work with Hans de Vreught and Joachim Trescher. They helped me a lot in finding and outlining a suitable research topic which I could use for my thesis. They carefully read draft versions of my reports, verified it in detail, and supported a good foundation for the subject of this thesis. When finishing my thesis, Hans still kept verifying the coherence of my work. It was great to work with them.

Not in the last place I want to mention the time and effort the commission members put into reading this book. The comments I received improved the clarity of presenting the thesis’ subject.

Thanks are also due to the group members Frits Kuijman, Kees van Reeuwijk, and in the last year Leon Aaronson and Marcel van Lohuizen. The working environment they evoked had certainly its impact on the contents of this thesis.

I loved the conversations I had with my friends which shaped my ideas on doing research, presenting research, and value research. Special thanks go to Gabriël Beckers and Daniël
Rutten, who wanted to be my ‘paranimfen’. The contents dependent cover design is credited to Huib van Nimmerdor. I certainly have not mentioned everyone whom I worked with, talked to, or read about, but I do appreciate their input. And finally, I say thanks to my dear girl-friend Anock Raes for her continuous enthusiasm and cheerfulness.

Paul Dechering
January 1998
Summary

Research in semantics and in compiler construction is often separated. The first has a theoretical character, in general, whereas the second has a more applied character. However, both fields are of fundamental importance in the interaction between a programming language and the behaviour of a computer.

The semantics of a programming language give a meaning to 'expressions' that can be made with this language. A compiler translates programs in a programming language to programs in another language on a, most often, lower abstraction level. Of course, the original meaning of a program must be preserved during the compilation process. So, there is a tight relationship between semantics and compiler construction.

This thesis focuses on high-level programming languages that are 'data parallel': that is, the specified data in a program can be used to direct the underlying system how to execute this program in parallel (meaning: doing a number of computations at the same time). This execution takes place in an SPMD model: Single Program Multiple Data model. As a result, a data parallel program can be executed in parallel, but does not need to. By this we mean that such a program initially has sequential semantics. Therefore, the first half of this thesis is spent mostly on the sequential semantics of a data parallel language.

In general, data parallel programs will have a parallel execution. Thus, somewhere during the compilation process the implicit directions for parallelism must be transformed to explicit parallel code. After presenting the sequential semantics we proceed with a closer look at certain language constructs that imply a kind of parallelism. We describe in detail the relation between the semantic definitions and their use in the compiler.

Finally, we present semantics for explicit parallel code: that is, there is not just a hint for parallelism, but the code explicitly prescribes a partition in the computation.

Apart from the interaction between semantics and compiler construction we are interested in a special kind of compiler: an optimizing compiler. This compiler is able to apply a set of optimizing transformations to a program. An advantage for these transformations is the availability of multiple levels of abstraction as described above. As such, transformations can be used to switch from data parallelism (implicit parallelism) to explicit parallelism, and, of course, to rewrite code on each abstraction level to obtain a more efficient program. Having a semantic model offers the ability to verify the correctness of these transformations.

The data parallel language we use for our semantic model and our compiler is called
V-nus. It is a newly defined intermediate language that can serve as a platform to which other data parallel languages can be translated. Once such a translation is made, one can make use of the V-nus compiler to experiment with optimizations via the transformations. Currently, V-nus is a mature language for which a semantic model, a compiler, and a set of transformations are available.
Chapter 1
Introduction

Oinos. But in this existence, I dreamed that I should be at once cognizant of all things, and thus at once be happy in being cognizant of all.
Agathos. Ah, not in knowledge is happiness, but in the acquisition of knowledge! In for ever knowing, we are for ever blessed; but to know all, were the curse of a fiend.

The Power of Words
Edgar Allan Poe

1.1 Semantics and compilers

Semantics and compilers are the two main keywords in this thesis. Both these keywords have a more general meaning than is used here. In general, semantics are 'the doctrine of meaning'. In Webster's New World Dictionary one can find a description that is more specific to our context: 'the relationships between signs and symbols and the concepts, feelings, etc. associated with them in the minds of their interpreters'. We use semantics to formalize the meaning of expressions in a specific programming language.

A compiler is described in (the same) Webster's dictionary as 'a person or thing that compiles; specif., a computer program that translates instructions, other programs, etc. in a high-level language into a machine language'. We leave it undefined what exactly a machine language is; the concept of a compiler as we use it is a computer program that translates programs in high-level languages to programs in low-level languages. High-level languages make it possible to express a certain computation in a more abstract context than low-level languages do.

So, when we have to deal with the translation of a program from one language to another, it is important to have a notion of the meaning of the original program, since a compiler should not change this meaning. From this point of view semantics and compilers are strongly interwoven. This thesis describes our research on the interaction between semantics and compilers.
1.2 Research question

Much research effort is put into semantics of programming languages and compiler construction (see the rest of this chapter). However, there is often a discrepancy between these topics. In case the focus is on semantics, the investigated programming language is not always extended enough for using it to program 'mature' programs. In case constructing a compiler is the main focus, one often lacks a formalization of the programming language concepts. Especially for imperative languages semantics and compiler construction are scarcely interwoven in practice.

The compiler we have in mind should not only make a correct translation, but should also be able to do optimizations in order to obtain more efficient code. The high-level language that is to be translated is a so-called data parallel language. In short this means the following (we come to this in more detail in Section 1.3). We have a computer with distributed memory (multiple memory parts). Consider a program in which it can be expressed that the data it uses is scattered over this distributed memory. A computation that uses a lot of data can then be split into sub-computations, each using data from only one memory part. Under certain conditions these sub-computations can be executed in parallel. As such, parallelism has originated from the data distribution directives, which explains the term.

The data parallel language Booster [76] [77] has been used in a first attempt to construct the described compiler. This language is especially suited to express operations on array-like data structures. In [44] a calculus (the View Calculus) is described in which transformations on Booster can be expressed. A special set of references to elements of such an array-like data structure, called a 'view', plays a major role in this calculus and Booster. A more tight relationship between the View Calculus and Booster is given in [77] where the View Calculus was labeled 'V-cal'. Unfortunately, efforts to construct a compiler that is able to translate all language constructs of Booster have failed. A possible barrier in this research project was the rather huge difference of abstraction between Booster and the final target code of the compiler.

Having the experience of this Booster project, the research question arose: is it possible to construct a compiler framework that translates a high-level data parallel language like Booster to a low-level language like C, in which it is possible to use a calculus of transformations? Of course, the translations and transformations should preserve the original meanings of programs written in the high-level language.

Considerations and decisions in answering this question are presented in Section 1.8.

1.3 Data parallelism versus other parallelism

We have introduced the term data parallelism with a short explanation. But since data parallel languages are the kind of languages that are input to our compiler, we need to look at this concept more closely. The term refers to a special kind of parallelism and suggests the existence of other forms of parallelism. Terms that can be found in literature
1.3. DATA PARALLELISM VERSUS OTHER PARALLELISM

are, for instance, ‘control parallelism’ (e.g. [50], [54], [57]), ‘task parallelism’ (e.g. [53], [62], [90]), ‘process parallelism’ (e.g. [50]), and ‘explicit parallelism’ (e.g. [54]). All these non-disjunctive forms of parallelism describe methods to express parallelism directly by using keywords of the programming language. In other words, the syntax of the programming language specifies which computations can be executed in parallel independently of the involved data, whereas parallelism of a data parallel program is dependent on the actual distribution of data over the available (distributed) memory.

It is not our intention to present a complete overview of possible forms of parallelism here. We will describe the concepts of data parallelism and these ‘other’ mentioned forms of parallelism, since these forms of parallelism are commonly used for imperative languages. We are aware of other forms of parallelism, like functional parallelism in functional languages, but we consider this to be outside the scope of this thesis.

Despite the widely accepted idea of the distinction between data parallelism and these other forms of parallelism, the terms are not always ‘defined’ as strict as one might assume. It seems that most people agree on the ‘definition’ for data parallelism. In [50] it is described as “working on a collection of data in parallel, at the same time”. In [62] it is said that “data parallelism arises from partitioning the data domains of a program and using the data partition to imply a partition of computation”. Another description is given in [90] stating that “data parallelism is typically expressed as a single thread of control operating on data sets distributed over all nodes”. But then, in [54] it is said that a data parallel language has a notion of explicit parallelism too. In using explicit parallelism it is not the data that is the primary cause for the introduction of parallelism. In this case, achieving parallelism through the simultaneous execution of the same operation across a set of data is the criterium for calling it data parallelism.

With the above descriptions of data parallelism it is ambiguous whether the following code example implies data parallelism or not. Consider a data structure $D$ that is distributed, and assume that the procedures $P$ and $Q$ do different computations.

\[
\text{IF}\ \text{owner}(D[0]) = \text{me} \ \text{THEN}
\begin{align*}
\text{call}\ P \\
\end{align*}
\text{ELSE}
\begin{align*}
\text{call}\ Q \\
\end{align*}
\text{ENDIF}
\]

According to [62] it should be called data parallel; with the description of [54] it is not data parallel since $P$ and $Q$ represent different operations.

By distinguishing data parallelism and control parallelism it is claimed that “control parallelism is the parallelism achievable through the simultaneous execution of different operations” [54]. In [57] the same concept is referred to by mentioning parallelism that “comes from multiple threads of control”. But in [50] a slightly different description is presented, asserting that “control parallelism is to exploit opportunities for parallelism in control statements”. There, process parallelism specifies “loosely coupled processes that (possibly) operate towards the same goal”.
CHAPTER 1. INTRODUCTION

Task parallelism is, in our opinion, used as a synonym for control parallelism. But then again, the descriptions remain ambiguous. “Task parallelism is typically expressed as a collection of sequential processes with explicit communication between them” [90]. A wider description is given in [62] where “task parallelism partitions the problem solution into functionally different tasks, each of which represents a different part of the computation”.

With this discussion we want to give an overview of some different forms of parallelism that are distinguished, and show that it is hard to give clear-cut descriptions. Furthermore, we want to introduce our own terms for referring to some kind of parallelism. In this thesis we use the terms ‘implicit parallelism’ and ‘explicit parallelism’. Implicit parallelism is present in language constructs that can be executed sequentially, but have potential for an execution in parallel. So, it depends on the ‘smartness’ of the compiler how much parallelism can be discovered, and thus whether implicit parallelism was present. In fact, one can conclude that if a compiler can produce parallel code for at least one program written in the input language, then the input language is implicitly parallel. Explicit parallelism is defined more concretely. Explicit parallelism requires a notion of a process (in the sense of a virtual computer; this is explained in more detail in Section 2.3.4). Strictly speaking, this can still be a single sequential execution trace, with the property that it can identify itself as a process. Only when a notion of a process is available – i.e. in the context of explicit parallelism – it makes sense to discuss language constructs that send packages of data to a process, or receive packages from a process.

1.4 Using data

The high-level languages we are interested in are data parallel languages. These languages are specially suited to express computations on large sets of data. In order to have the data available in a program of such a language the data must be accessible via some mechanism. The kind of applications we are looking at typically perform operations on vectors, matrices, and the like. For this reason our main data structure is an array-like structure. An array-like data structure is a ‘rectangular’ structure consisting of data, where each data element is identified with a coordinate or, as we call it, an index. For example, consider the matrix $A$:

$$
A = \begin{bmatrix}
a & b & c \\
d & e & f \\
g & h & i
\end{bmatrix}
$$

We can record the matrix $A$ in a program variable $A$ in pseudo-code as follows:

$A := [[a, b, c], [d, e, f], [g, h, i]]$

Note: the notation has no implications for the kind of implementation. In mathematics, the element $c$ in matrix $A$ can be referred to with $A_{(0,2)}$ (we count from zero); in this pseudo-code we would use the expression $A[0,2]$. The pair $(0,2)$ is called the index of $c$ in $A$. Other forms of recording such data are possible, but for the kind of operations
we want to perform on the data, an array-like data structure is most suited. It is called
rectangular since the sequences of elements in one dimension are of equal length. The
advantage is that boundary checks for accessing array elements are simple and can be
implemented efficiently. The disadvantage, however, is unnecessary use of memory space
in case sequences of elements in one dimension do not need to be of the same length (e.g.
sparse matrices).

There is a small but essential difference in looking at the ‘dimensionality’ of an array.
One can think of the matrix A as a 2-dimensional structure, in which the elements a, b,
..., i are placed. However, considering the program variable A one might think of it as
being a 1-dimensional structure consisting of the elements [a, b, c], [d, e, f], and [g, h, i].
In the first case A[1] is undefined, in the second case A[1] represents a vector. For our
research, we will only consider array-like structures that consist of scalars (flat arrays).
Instead of arrays that also may consist of arrays (nested arrays); e.g. the matrix above is,
in our terms, 2-dimensional (see also Section 1.8).

Although a reference to an element in an array always refers to a scalar, it is not
simply another way of denoting a scalar. Arrays make it harder to give a meaning to
certain operations. In [38] the difference between ‘scalar notation’ and ‘array notation’ is
presented with a parallel assignment example. The parallel assignment \( x,y := y,x \) is valid,
and means assigning \( y \) to \( x \) and ‘at the same time’ assigning \( x \) to \( y \). As a result, \( x \) has the
old value of \( y \). and \( y \) has the old value of \( x \). In this context, the assignment \( x,x := 1,2 \) leads
to an error. But how about the assignment \( A[i], A[j] := 1,2 \)? It leads to an error when \( i \)
represents the same value as \( j \) does, but in general that is only known at run-time. In case
of the scalar notation one can deduce such an error at compile time.

In [6] another example is given in which it is shown that one has to be careful in giving
a meaning to array operations. We follow [6] with the following examples. It is trivial that
the assignment (scalar notation): \( x := 1 \) means that, after execution, \( x \) represents the value
1 (we use the shortcut \( x = 1 \)). In the same way, one might expect that the assignment
(array notation): \( A[i] := 1 \) means that \( A[i] = 1 \) after execution. This is not true when, for
assignment can then also be written as \( A[A[2]] := 1 \), such that, after execution, we have
\( A[A[2]] = 2 \).

Nowadays, many languages exist using array-like data structures, many of them having
their own solution to the difficulties that arrays may cause. e.g. FORTRAN90 [59], HPF [41],
APL [60], VCODE [15], NESL [12], ZPL [66]. Except VCODE these languages offer the user
language constructs that ease the task of programming with arrays.

The language APL was defined by K. Iverson in 1962 [60] and started as a blackboard
notation. In 1966 it was implemented on a computer at the IBM T.J. Watson Research
Center. Today, the language is still used as input for current compilers, and is suited to
direct the construction of compilers that exploit data parallelism in APL programs based
on parallel high-level primitives [24]. ‘Classical APL’ uses flat arrays and has a wide set
of high-level operators on arrays available. It is an attractive research vehicle for studying
the effectiveness of various proposed compilation techniques aimed at exploiting implicit
parallelism.
FORTRAN90 enhances the FORTRAN77 language by, among other features, the introduction of operations that operate on whole arrays or sections of an array. Instead of constructing loops in order to express a computation for each element of an array, FORTRAN90 offers special language constructs for such a computation. Programs written using these array operations are often clearer and are more easily optimized by compilers than FORTRAN77 programs. These new language constructs are also more suited to find potential parallelism, since the semantics of the constructs can be used more effectively than the semantics hidden in a user-constructed loop.

In FORTRAN95 [40] and High Performance FORTRAN (HPF) [41] the programmer is enabled to explicitly express a kind of independency of computations over an array, using the forall keyword (see also Chapter 5). This in contrast with the previous FORTRAN language instances where the compiler should find out how to exploit parallelism. Furthermore, in HPF the used data of the program can be distributed over the involved processes by the data distribution declarations. Next to these declarations, imperative language constructs can be used to redistribute data.

When addressing elements in an array the previous languages use numbers (integers) to explicitly appoint specific array elements. Another way of using arrays on a high level in computations is defined in the language ZPL [66], using implicit references to array elements. Apart from claiming ease of programming and clarity, it is also claimed that it leads to efficient parallel execution. In an example the 4-point Jacobi computation is considered where each value in the array is to be replaced by the average of its four nearest neighbours. The computation for an arbitrary element in the array can be given in terms of ‘east’, ‘south’, . . . relative to this element, as such appointing the neighbours. With similar constructs one can set boundary conditions for complete regions of an array.

NESL is a functional data parallel language [12]. The main data structure in this language is called a ‘sequence’: a wide set of parallel operations on sequences is available. A sequence is an ordered set, and may contain other sequences of different lengths. In this way one can construct a non-rectangular data structure, which is especially suited for sparse matrices. The compiler for NESL generates VCODE, a data parallel intermediate language [15]. VCODE contains a small set of stack-based instructions, most of which operate on one-dimensional vectors of atomic values. It is an intermediate language because it lacks many features associated with high-level languages, but has the mechanisms to implement such features.

1.5 Towards a compiler framework

So far we have introduced the concepts of compilers, semantics, and (data) parallelism. We will now take a more in-depth look at the approach we have taken to answer our research question. Below we describe how the mentioned concepts are embedded in our compiler framework.

Traditional compilers for data parallel languages advocate the ‘one tool does all’ approach: parsing, optimizing, and code generation are strongly interwoven and they are
1.5. **TOWARDS A COMPILER FRAMEWORK**

often hard-coded in the compiler. For our compiler we have taken a different approach where all these phases are handled by separate tools.

There are several advantages of this approach over the 'one tool does all' approach:

- The different tools are easier to develop, to maintain, and to extend because they do just a single job. There is no interweaving with other tools.

- It is easier to replace the individual tools. A change in the front-end means that a different language can be translated. Choosing a different target machine will mean that the back-end must be changed.

The disadvantage of our approach is a little drop in speed of the compiler due to the overhead of running several tools. Each tool has to parse its input and generate its output. In our opinion the flexibility of the new approach outweighs the minor decrease in speed.

![Diagram](image)

Figure 1.1: Overview of the V-nus compiler

Our experimental compiler consists of three tools (see Figure 1.1) [91]: a front-end that parses the input program, an optimizer that operates on a intermediate formalism, and a back-end that generates target code. For the input program we focus on array-like programming languages, like *Booster* and *FORTRAN*. We have chosen to define a new language as the intermediate formalism, called *V-nus* (*V-cal* Normal-form of Unified Statements), such that we could choose our own level of abstraction, and modify the language when desirable. The target code is C++ (possibly extended by parallelizing libraries).

The front-end only parses the input program and checks the static semantics of the program. The front-end does not attempt to do any optimizations. The output of the front-end is a program in our language *V-nus*.

Since *V-nus* is the interface between all tools, we have several requirements for such an intermediate language:

- *V-nus* is self contained. A *V-nus* program is the only information that is exchanged between the different tools.
• *V-nus* must be suitable for the definition of a calculus of transformations.

• *V-nus* must be expressive enough to describe all constructs of the input language.

• Since all tools must parse and/or generate *V-nus* code, the language must be extremely simple such that only little time is lost while parsing and generating *V-nus* code.

We have redefined the calculus *V-cal* (in relation to [77]) on *V-nus* programs that tries to improve the performance of *V-nus* programs. The calculus consists of transformation rules that are to be applied by an engine to the nodes of the parse tree of a *V-nus* program (see the left part of Figure 1.2). The conditions under which the application of a transformation rule is allowed, can depend on tree patterns, data flow analysis, and heuristics. Both input and resulting output are valid *V-nus* programs.

![Diagram](image)

(a) *V-nus* grammar  
(b) *V-cal* rules  
(c) *V-cal* engine

Figure 1.2: *V-cal* engine

As an advantage of this approach the optimizing engine can be implemented by a set of specific tools. As illustrated by the blow up in the right part of Figure 1.2 each of these tools reads a *V-nus* program, performs a particular type of optimization, and then generates a modified *V-nus* program. The final *V-nus* program resulting from this process will then form the input of the back-end. Its job is to make a direct translation from *V-nus* to the target code. Just like the front-end it does not do any optimizations.

### 1.6 Intermediate formalisms

We present an overview of different approaches to intermediate formalisms that streamline data parallel compilers.
1.6. INTERMEDIATE FORMALISMS

Research on compilers that generate distributed code from sequential or data parallel algorithms concentrates on problems in the area of scientific computing, where shared arrays have to be distributed among the private memories of different machines. In particular methods that extend FORTRAN [59] by array constructs and means to specify a decomposition of an array have been studied extensively [19] [4] [58] [102].

While these approaches promise immediate advances of the state of practice, it is also acknowledged that many algorithms described in a few pages of mathematics often result in large FORTRAN programs which make it difficult to experiment with simple variations of the algorithm, or to port such a program to a variety of architectures [85]. Several experimental approaches have been proposed that address these problems by more fundamental algorithmic, semantical, or linguistic considerations [61] [56] [70].

One characteristic of these approaches is to extend the usual notion of array types to provide arrays with more general index sets, e.g. maps in Fidil, selective assignments in F-Code, Psi Reductions in the $\psi$-calculus, and the view concept of Booster [76] and V-nus. The second characteristic these intermediate formalisms have in common is a formal approach to the definition of the language semantics and compilation process.

F-Code [61] is a formally defined language that was developed to represent the semantics of data parallel processing, as well as data management and control primitives of imperative languages. F-Code was designed as an intermediate language for compilers to act as a portable software platform with the purpose to foster an architecture independent programming model. In contrast to our work on V-nus that tries to facilitate research on novel programming paradigms and associated compiler technology, F-Code uses only one level of abstraction which implies less diversity in possible program transformations.

Like V-nus, Fidil [56] extends the semantic domain of FORTRAN-like algebraic languages with facilities for construction, composition, and refinement of index sets –called domains– and for performing computations on functions defined over these domains –called maps. Additionally, it provides capabilities for defining operators and for performing higher order operations on functions.

Fidil is an attempt to automate much of the routine book-keeping that forms a large part of many programs, and to bring the semantic level of these programs closer to that at which the algorithms are originally conceived. However, the versatile nature of maps, like that of views in V-nus, is a potential source of inefficiency. A set of techniques to reduce or eliminate maps have been described in [85], and it would be interesting to obtain performance figures on realistic programs that show evaluations of these techniques.

In [70] a calculus on arrays, called the $\psi$-calculus, is given that is capable of transforming a high level single massively parallel operation on arrays into a low level version optimized for a given parallel architecture. The $\psi$-calculus is just like V-nus based on operations acting on the indexing scheme of arrays. Given a data structure, a communication pattern, and a topology of the parallel machine the $\psi$-calculus is able to compute an efficient memory mapping with respect to some cost function. For V-nus this information is in general not known during compile time. In some cases we will need to resort to default parameters to compute a mapping. The $\psi$-calculus could be embedded as an auxiliary calculus at the lowest level of the compilation process of V-nus. In such a framework V-nus would be used
CHAPTER 1. INTRODUCTION

for the higher level constructs (like control flow) and establishing the parameters needed for the ψ-calculus, which in turn could compute a good mapping scheme.

Also in [83] it is recognized that denotational semantics are seldom used as a specification formalism for compiler construction. An empirical study is described of compiler generation for data parallel languages from formal specifications based on denotational semantics. As such, the primary goal was to investigate how to derive a practical compiler from these specifications, rather than constructing an optimizing compiler. In our compiler, the denotational semantics are used to define the intermediate language on its different abstraction levels, and can be used as a separate verification system for optimizations. In [83] the denotational semantics are an integral part of the compiler constructor.

1.7 Semantics

Semantics for programming languages are used to have a hold on the meanings of programs. Nevertheless, to express the meaning of a program one has to rely on the assumption that the meaning is known of the formalism that is used for expressing this meaning. For instance, one can choose to rely on mathematics to formalize 'the meaning' of a program. This gives cause for a second consideration: what is the meaning of a program? This highly depends on the interest one has in a certain program (or programming language). If, for instance, one is only interested in the outcome of the computation that is programmed, it does not matter how the computation is done, but only what the outcome of the computation is. Conversely, when time is of importance too, it may be significant to have a difference in meaning when two computations differ in their sequence of actions, but still have the same computed result. A simple and concrete example that reflects these considerations is the program fragment: x := 1; x := 1. One can both think of this fragment meaning the same as x := 1 and think of these fragments having a different meaning.

Different views on the concept 'meaning of a program' and different formalisms for expressing this concept have caused the existence of different semantic models. One can find semantics called operational semantics, denotational semantics, axiomatic semantics, algebraic semantics, action semantics, semantics of assertions, and so forth [6] [68] [88] [98]. We will not give an overview of all these semantics, but we spend a few words on the operational semantics and the denotational semantics.

The purpose of operational semantics is to describe how a computation is performed. As such, the operational semantics reflect the computation steps taken by a computer when it is executing a certain program. So, the two program fragments above (expressing two assignments and one assignment respectively) are distinguished. Having operational semantics of a programming language is often an assurance that this language can be implemented; i.e. it is possible to construct a compiler.

Denotational semantics are based on the idea that language constructs denote certain values. Most often, these values are expressed as mathematical objects, and this explains why these semantics are originally called mathematical semantics. Quoting from [6], a characteristic of denotational semantics is "the application of various mappings from the
1.7. SEMANTICS

entities in the linguistic world to appropriate objects in the mathematical world in such a way that the meaning, or value, of a construct is determined in terms of the meanings of its constituent components'. That claims that the meaning of a program can be given by composing the meanings of its parts; which accounts for calling them compositional semantics as well. A major advantage of having compositional semantics in combination with a calculus of transformations on the programming language, is the ability to give semantics to parts of a program instead of requiring the complete program. One can focus on only the part of the program that is affected by a transformation rule, and see whether the transformation preserves the semantics.

Since denotational semantics do not reflect the actual computation sequence gone through by a computer in general, there is no guarantee that the language in question can be implemented.

In this thesis we use denotational semantics to give a meaning to the language V-mus. Denotational semantics are preferred over the operational semantics because of their use for a calculus of transformations. Other compositional semantic models (abstract semantics, algebraic semantics, et cetera) might have been chosen as well, but in our opinion there was no additional advantage for these other models with respect to the denotational semantics.

Among the first researchers formalizing the semantics of programming languages are D. Scott and C. Strachey [84]. They note that the concepts of the original language may be lost or at least obscured during the translation of programs into machine language. Having semantics may prevent this; it is shown how the constructed mathematical semantics can be used to define equivalence of programs.

Research on semantics is extended and presented in more detail in [6]. A notion of partial correctness of a program is defined, making use of the semantics of assertions. Proof systems are presented for the formulae constructed with the assertions. It is shown in full detail what the relation is between these semantics of assertions, the given operational semantics, and the given denotational semantics.

Also in [68] the reader is introduced to different forms of semantics. With respect to [6] the focus here is more on practical programming problems. A wide set of program fragments and their semantic representations is given.

An in depth 'introduction' to the theory of semantics is given in [98]. Following a presentation of domain theory, the semantics and methods of proof for several functional languages are treated. Programming aspects like call-by-value and call-by-name evaluation, eager and lazy evaluation, and parallelism are described.

A broad overview of different semantics is given in [88]. Based on small exercises it is presented how programming constructs are represented in the different models for semantics.

A wide spectrum of control flow notions can be found in [8]. The meanings of these language constructs are given with a presentation of semantics of 27 languages. It gives an overview of how to construct denotational and operational semantics for all these kinds of control flow.
1.8 Research question revisited

The approach taken to answer the research question, mentioned in Section 1.2, is to construct a concise intermediate language, called V-nus, that is defined by formal semantics. In [79] it is recognized that a trade-off exists between offering high-level semantics and generating efficient code in a single level of abstraction. Therefore, V-nus will not serve as a language platform with a single level of abstraction, but instead will offer multiple levels of abstraction. In [81] it is pointed out that programmers are often provided a conventional programming language, enhanced with a few low-level constructs for creating and synchronizing parallel processes. In our approach each abstraction level will be fully capable of representing the original program, without making use of language constructs of a different abstraction level.

Transforming a V-nus program from one abstraction level to another will be done by transformation rules of the calculus V-cal. With this approach we will build a modular compiler framework that is more orderly than traditional compilers (with the ‘one-tool-does-all approach’). Note that this is an engineering objective for compiler construction.

The semantic model for V-nus will serve several purposes. In constructing V-nus we will start from investigating language concepts needed for an intermediate formalism for the class of input languages mentioned. The language V-nus can then be constructed in tight coherence with the construction of well-defined semantics for these language concepts. Furthermore, as mentioned earlier, the semantics will be used to verify the transformation rules of V-cal. This requirement explains why we have chosen to use denotational semantics (compositional semantics). Since these denotational semantics will, in fact, define the meaning of the language V-nus, there is little need to construct operational semantics as well. Operational semantics might show that the construction of a compiler for V-nus is possible indeed, but we use two other arguments to convince the reader that such a compiler is possible. Being ahead of Chapter 8 we can claim to have a compiler for V-nus. Secondly, for the highest level of abstraction we will construct semantics based on functions that are close to a functional program. Such a functional program would be a special kind of a compiler.

The semantics we are about to construct is based on the input/output relation of computations. That is, we record the states a ‘computer’ is in before and after performing a computation, and this pair of states represents the meaning of this computation. So, intermediate results (computation traces) are of no interest. The use of semantics in combination with the transformation rules justifies this choice. For the verification of a transformation rule we only need to know whether the ‘end state’ remains the same for a given ‘begin state’.

In our research question we focus on input languages that are imperative and data parallel. However, a certain amount of our language V-nus will offer declarative language constructs as well. These are typically used for the specification of the data distribution, program variable type information et cetera. This thesis will not investigate and solve problems encountered in semantics on declarations. A simple unorthodox strategy will be chosen to cope with these declarations.
In the context of this thesis, we are not specifically interested in the ‘contents’ of the calculus V-cal. That means that we do not concentrate on what the transformations express, but instead concentrate on how transformations can be expressed and used in our compiler. For further reading on these transformations we refer to [17].

1.9 Thesis overview

Each chapter has in its introduction, if necessary, a paragraph on the notations and concepts that are new, and are used in that chapter. Each chapter assumes knowledge of the previous chapters, except Chapter 7 which basically can be read on its own. A basic knowledge of semantics and compilers is needed for reading this thesis.

Most definitions given in this thesis are preceded with an introduction to the idea (and usage) of it. The formal definition is then succeeded with a description of the technical details.

For some chapters further investigation on literature is presented in one of its first sections.

Chapter 2. In this chapter the intermediate language V-nus is introduced in more detail. We investigate its role in our compiler, and put into words the requirements we have for such an intermediate language. Thereafter, we describe the different language constructs and give their informal semantics. Parts of this chapter have been adopted from [18].

Chapter 3. The concept of a semantic state is introduced here. Several semantic states are defined which will be used to define the semantics of V-nus. These states are: the declaration state, the index state, and the program state. The declaration state is needed to use the declarations of a V-nus program. The index state represents the current values of index variables, where index variables are typically used to specify the index space of loops. The program state records the values of the ‘global’ program variables. It is shown that the set of possible program states is well-defined. Parts of this chapter have been published in [28] in which a former model of semantic states is presented.

Chapter 4. Here, the semantic states are used to define the meanings of the program constructs of V-nus. These meanings are given in denotational semantics. We start with some requirements for semantic states when these are used initially. Thereafter, all language constructs used for sequential execution only will get their meaning. This chapter, too, describes work that has partly been published in [28].

Chapter 5. The results of our research on implicit parallelism are presented in this chapter. It is described how implicitly parallel language constructs fit in V-nus, and how these may be adopted in higher-level languages. The semantics of these language constructs can still be defined in terms of the semantic states used so far. Most topics of this chapter have been published in [29] [30] [31] [32] [33].
Chapter 6. This chapter covers the topic of explicit parallelism. By (our) definition this requires the concept of a process. As a consequence an extension to the semantic states is needed. Two levels for defining semantics are presented: one for the meaning of a single process, and one for the meaning of 'all' processes together.

Chapter 7. The calculus V-cal is explained in more detail. We give examples of the kind of transformations we are interested in. We present a structure in which these transformations fit and which is suited to be used as an optimizing engine for our compiler. This chapter is based on [35] [36].

Chapter 8. We end our thesis with a short discussion on the presented research topics, where we draw our conclusions and mention some future work. We once more discuss our research question.

Appendix A. One can find the complete context free grammar of the language V-nus in this appendix. We often use language constructs of V-nus by referring to non-terminals as these are named in this appendix. Furthermore, a basic description is given on the different items that belong to the symbol table of V-nus. A few words are spent on the requirements of an ASCII file representing a V-nus program.

Appendix B. Several (semantic) functions are introduced stepwise in this thesis. As such, these functions are defined by different parts given throughout the text. One can look up the complete definition of these functions in this appendix. No extras are added in this appendix; it is simply there to give the reader an easy reference to these functions.

Appendix C. One of the implicit parallel language constructs, presented in Chapter 5, is the for all term. In different languages this for all term may have different semantics. This appendix puts several examples of for all terms together, and gives their semantics in the contexts of different languages.

Appendix D. This appendix gives a short presentation of using the semantics for proving the correctness of some transformation rules in our optimizing engine: V-cal. Two transformation rules are considered of which is shown that these are semantics preserving when applied to a V-nus program.

Appendix E. Here, a list of used symbols is presented with a short description/hint of its function, together with a reference to the page where it is first introduced.
Chapter 2

V-nus and its informal semantics

"Be what you would seem to be" – or if you'd like it put more simply – 'Never imagine yourself not to be otherwise than what it might appear to others that what you were or might have been was not otherwise than what you had been would have appeared to them to be otherwise".

Alice's Adventures in Wonderland
Lewis Carroll

2.1 The idea behind V-nus

V-nus is a language that is constructed to serve as an intermediate representation in a compiler framework for data parallel languages. The framework we use consists of three parts: a front-end that translates a high-level programming language to the intermediate language V-nus, an optimizing engine that rewrites V-nus code, and a back-end that translates V-nus code to target code. The central question in constructing the V-nus language was: for what compile phases should it be suited? In the literature one can find several language concepts intermediate languages can be judged on. We list a few of them, referring to the kind of intermediate language we are interested in, with 'the language'.

- The language should have high-level semantics, including all sorts of abstractions inherent in high-level languages [50] [61] [86]. It should be useful for constructing highly parallel programs [66] [67].

- The language should have a primitive syntax such that it can be easily parsed by the compiler [15] [61]. Combined with this, it needs descriptive simplicity [66] [86].

- The language must be independent of the underlying architecture(s) [61] [67] [86].

- It should be possible to generate efficient code [15] [61] [66] [67].
Eventually, the question has been answered with a list of requirements for \textit{V-nus} which we will discuss now.

\textbf{Program transformations} One of the main features of \textit{V-nus} is to allow for program transformations. Once a program is translated to the \textit{V-nus} language, program transformations can be used for optimizations.

\textbf{High-level semantics} \textit{V-nus} must be able to express high-level semantics. That means that \textit{V-nus} has (almost) the same descriptive power as the original language that is processed by the front-end. Hence, information (for instance, use of recursion or iteration) about the computation described in the original program is preserved when it is translated to \textit{V-nus}. Then, program transformations can be more powerful since high-level semantics can be used within the intermediate representation. Another important consideration that requires high-level semantics is their use for existing libraries of templates. A template is a representation for a rewrite rule that matches on code patterns. As early as possible in the compilation process program constructs can be recognized that are also present in libraries of templates [64]. When the original program has been unraveled too much, certain code patterns can not be recognized anymore as an instance of an existing code pattern.

As a result, having high-level semantics in \textit{V-nus}, a rather straightforward front-end can be used in the compiler framework.

\textbf{Low-level semantics} Once the original program is expressed in \textit{V-nus} without loss of information, it must be possible to rewrite the program into lower-level program constructs. Important transformations are those that convert a data parallel program to an explicitly parallel program; i.e. a program with explicit communication primitives. These and other transformations are carried out by optimizing engines. Mirroring the advantage of the high-level semantics, the low-level semantics do not require a complex back-end.

\textbf{Generality} Only a single language should be used for compiling and optimizing programs of different high-level, imperative, data parallel languages. All effort can be put into the engines that transform \textit{V-nus} programs. For an arbitrary data parallel language it should suffice to build a front-end that translates to \textit{V-nus}.

\textbf{Semantic framework} Rigorous semantics are required to serve as a base for transformation rules. Since the meaning of \textit{V-nus} programs can be expressed formally one can verify whether a transformation preserves the semantics of the original program.

The class of languages for which \textit{V-nus} will serve as an intermediate language consists of high-level data parallel languages. Typically, these languages use mechanisms to express implicit parallelism. The main data structure that is used for this purpose is the array structure. It is carefully studied how these data structures can be used efficiently. That means that (1) manipulating indexes of the arrays is well embedded, and (2) typical operators of the data parallel domain are present in \textit{V-nus}. Examples of these typical
operators are the ‘reduce’ and the ‘scan’. The reduce combines elements of an array into a scalar; for instance by summing these elements. The scan operator is a generalization of the reduce operator. It combines elements of an array into another array. Other typical control structures present in data parallel languages, like a ‘forall’, are also present in the language. All these language constructs imply some form of parallelism that depend on the distribution of data, expressed by data distribution directives.

2.2 Definitions

V-nus programs specify computations that act on components called locations. Variables are bounded sets of locations that represent a value determined by the variable’s type. Variables represent a data structure. If a value $E$ can be expressed by a variable of type $T$, then we say the value is of type $T$. In this thesis we use the types ‘scalar’, ‘shape’, and ‘view’ for data structures in V-nus. Their meaning will be explained later. For a formal description of these types we refer to [18]. It is described in the context of the language Booster, but the same type specification is used in V-nus.

The rank of a data structure is the sequence of values of its length in each dimension. The rank of a zero dimensional data structure (i.e. a scalar) is the empty sequence.

An expression specifies a computation that produces a value or a data structure.

2.2.1 Notational conventions

For the rest of this thesis we adopt the following notational conventions:

- Let $N$ be a nonterminal of the V-nus grammar. The term $\mathcal{L}(N)$ denotes the language defined by the production rules of the V-nus grammar and the start symbol $N$. We say that $\mathcal{L}(N)$ is the class of $N$.

- Let $N$ range over all nonterminals of the V-nus grammar. The set of all V-nus program (fragments) $\mathbb{V}_{\text{nus}}$ is then described by: $\mathbb{V}_{\text{nus}} = \bigcup_N \mathcal{L}(N)$

- For the representation of V-nus program fragments we will use constructs of the extended BNF defined in Appendix A in an obvious way. Identifiers in italics denote the language construct suggested by the V-nus grammar. Identifiers in sans serif denote a terminal of the V-nus language. For instance, in the program fragment

  \[
  \text{view Id View}
  \]

  the term view is a terminal. The identifier Id is an element of $\mathcal{L}($IDENTIFIER$)$; View is an element of $\mathcal{L}($View$)$.

- The value of a V-nus expression $e$ is represented by the same syntax in italic. In order to make a clear difference between operators of the V-nus language and their look-alikes in mathematics, we write a tilde above the operator to explicitly specify that the mathematical operator is meant. For instance, the value of the V-nus expression:
can be given by the mathematical expression $3 \times n$. Here, $n$ represents the value the variable $n$ has when it is used.

### 2.2.2 Representation

The representation of V-nus terms as sequences of characters is defined using the ASCII set. Terms are identifiers, numbers, strings, operators, and delimiters. The following lexical rules must be observed: blanks and line breaks must not occur within terms. They are ignored unless they are essential to separate two consecutive terms. Upper-case and lower-case characters are considered as distinct.

### 2.2.3 Symbol table

Every identifier occurring in a program must have a declaration in the symbol table. Declarations specify certain properties of an identifier, such as whether it is a constant, a type, a variable, etc. The identifier is used to refer to the associated entity.

The scope of an identifier is the block (procedure, function, or statement) to which the declaration belongs and to which the identifier is local. Scopes can be nested. An identifier $Id$ has a defined meaning in a scope $S$ if a declaration for $Id$ in scope $S$ exists.

### 2.3 Program constructs

In this section we explain the different program constructs as these are defined in the V-nus grammar in Appendix A. It is not the intention to explain every language construct; only those constructs are mentioned that may need additional information about the syntax and proposed meaning.

#### 2.3.1 Units

A complete V-nus program is determined by the start symbol Production. Such a program consists of an initialization part, a declaration part, a list of statements, and a symbol table. Since languages that translate to V-nus may contain modules [18, 99], an initialization part has been constructed to represent all modules that are not a start module. Functions and procedures are defined in the declaration part. The list of statements defines the operational behaviour of the program. Finally, the symbol table is present, so that a V-nus program is self contained.

For the terms Statements and Statement it is possible to specify a Label. This is necessary when a reference is needed to a specific statement (or list of statements). Data dependency analysis needs to refer to these program constructs. In future versions of V-nus the scoping of variables may be extended such that the scope is defined in terms of these labels.
2.3. PROGRAM CONSTRUCTS

A list of pragmas can be part of a statement. Pragmas are annotations that are used to specify additional information to the statement, such that the compiler is aided in compiling it. Instances of pragmas can be found in Chapter 5 where we will show how pragmas are used in order to gain more efficiency when compiling the Forall statement.

It is possible to create nested lists of statements. For now, a nested list of statements $L_1$ has the same meaning as a flat list of statements $L_2$ where the order of statements of $L_2$ is equal to a depth first order of the statements of $L_1$. For instance, the nested statement list:

$$\text{statements } [s_1, \text{statements } [s_2, s_3], s_4]$$

has the same meaning as the flat statement list:

$$\text{statements } [s_1, s_2, s_3, s_4]$$

where the $s_i$ are arbitrary statements. Future versions of V-nus may extend the scoping mechanism, and then blocks of statements can be used to delimit a scope.

Functions and procedures are different program constructs. A function in V-nus is referentially transparent. We follow [39] in the meaning of referential transparency: a referentially transparent expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of the program to share the expression. All references to the value are therefore equivalent to the value itself and the fact that the expression may be referred to from other parts of the program is of no concern.

So, referential transparency requires the function body to use locally declared variables and actual parameters for data access only. A function call belongs to the class of Expression. A procedure need not to be referentially transparent. A procedure call belongs to the class of Action.

2.3.2 Declarations

A formal parameter of a function or procedure is represented by an identifier. The parameter is always passed by value. More detailed explanation on passing parameters is given in Section 4.5.1.

A cardinality specification is used to declare an index variable that is local to the statement in which the specification was given. A cardinality specification of the form: $(Id, E)$ declares a variable $Id$ with a range between 0 and $E - 1$ inclusive.
Example 2.3.1 Values for an index variable using a cardinality specification.
Consider a loop construct in V-nus:

\[
\text{iteration } [(i,5)] \text{ statements } [Stms]
\]

where the cardinality specification is \((i,5)\). This means that the index variable \(i\) will range over the values \(0 \ldots 4\). \(Stms\) will therefore be executed 5 times, and each time \(i\) represents one of the values 0, 1, 2, 3, or 4 (in this order).

2.3.3 Flow of control

For the terms While, Iteration, Forall, and Foreach we use a notion of a body instance. A body instance is equal to the body Statements where a specific value of the index variable is substituted for each occurrence of the index variable.

The non-terminals While, Iteration, and If represent language constructs with a straightforward meaning. The While term repeats as long as the given condition is true. The Iteration term has a similar meaning as the well-known for statement. It specifies an index space and accompanying body instances, which are executed in a predefined order. The If term executes the first given statement list if the condition is true, otherwise the second statement list is executed. The Forall term also specifies an index space and accompanying body instances, but the execution order is irrelevant. All body instances start in the same context, each executed once. For a detailed explanation of this construct we refer to Chapter 5. The Forall semantics very much resembles those of the Iteration, except that there is no predefined order; any order is fine.

Note that the index variable specified in the cardinality list has always a range that starts with zero and has a positive upper bound. By using program transformations one can easily rewrite loops with different loop bounds to loops that can be represented in V-nus (see [102]).

2.3.4 Communication

Communication statements come into existence when a data parallel program is transformed to an explicit parallel program. This process can be done in several steps. In order to express intermediate forms during this process the set of communication statements is not orthogonal.

A data parallel program expresses parallelism using the distribution information of its data. This way of specifying parallelism is called an 'implicit parallel' specification. It heavily depends on the sophistication of the compiler how an implicit parallel specification is transformed to an explicit parallel specification. An explicit parallel specification uses
language constructs that explicitly direct the compiler what code needs to be generated in order to involve parallel executions. In short: implicit parallel programs abstract more from describing parallel executions than explicit parallel programs do. There is, however, no clear line to be drawn between these kinds of parallel programs, for different intermediate forms are possible too.

When a program is written to be executed in parallel, the notion of a process will be used. We regard a process as a separate, virtual, computation processing unit (CPU) with memory that can be accessed. This memory can be local to the CPU or be part of shared memory. A process can execute programs on its own.

Explicit parallel language constructs on a rather high level of abstraction are want, synchronize, and redistribute. The statement want \( H \ E \) denotes that the expression \( E \) is needed for the execution of the statement indicated by \( H \). Typically, the expression \( E \) represents a data structure. As a consequence \( E \) will be sent to the process that will execute \( H \). synchronize \( H \ E \) means that the copy of data structure \( E \) has been changed by statement \( H \). So, a synchronization on \( E \) may be needed after execution of \( H \). Program execution of the process that encountered the synchronize is stalled until the owner of \( E \) has been updated. When a redistribution of data is needed, the statement redistribute \( Id \) can be used. Note that only the whole data structure referenced by \( Id \) can be redistributed. If parts of \( Id \) could be redistributed separately, the distribution table of a single data structure might become too large to be used efficiently. Normally, a distribution function describes the way a data structure is distributed. When only parts of a data structure are redistributed, different function descriptions need to be recorded for these parts. This enlarges the distribution table, which makes access to data structures less efficient.

The statement barrier is a generalization of synchronize. The process that executes a barrier will wait until all other processes have executed that barrier also. So, it is a synchronization on all data structures. It is the responsibility of the programmer to cause all processes to execute the same barrier at a certain point in time. Otherwise, a deadlock or a synchronization on different barriers can occur.

The statements send, receive, and broadcast are the most primitive communication statements (the least abstract denotation for parallelism) in V-nas. send From To Exp means that process From will send the expression Exp to process To. The Send is non-blocking; i.e., the process that sends does not need to wait for an acknowledgement. As such, asynchronous communication is reflected by send. receive To From Exp means that process To will receive the expression Exp from process From. Exp may only be an identifier which will represent the received data. The Receive is blocking; i.e., if a Receive is encountered by a process, execution of this process only proceeds when something has actually been received. Since the Send is non-blocking, the interaction between the process that sends and the process that receives is said to adhere to asynchronous communication\(^1\).

The statement broadcast From Exp expresses that the expression Exp will be sent by

\(^{1}\)The difference between synchronous and asynchronous communication is not clear-cut. Different points on the traveled path of the sent data can serve as synchronization points, which may cause a blocked process to proceed at different points in time. For more on the ambiguity of this subject we refer to [89].
process From to all other processes. In all three communication statements the source is present from which is sent (in case of the Send and Broadcast) or to which is received (in case of the Receive). This is done in order to use communication libraries in the compiler that need this information; furthermore, it is also convenient to have this information available in the communication statements for our denotational semantics as will be clear in Chapter 6. In Example 2.3.2 an expression is sent by one process and received by another process.

**Example 2.3.2** A Send - Receive pair.
Consider the following program fragment for a process identified as p1.

```
assignment (x[ ]) 3,
send p1 p2 (x[ ])
```

The first statement assigns the value 3 to the expression \(x[ \] \). The second statement causes the process p1 to send the expression \(x[ \] \) to a process identified as p2. Assume that p2 will execute the following code in parallel with p1.

```
assignment (x[ ]) 5,
receive p2 p1 x
```

When both processes have executed the presented statements, the value of \(x[ \] \) is 3 for both processes.

### 2.3.5 Statements

A special kind of statement present in the \(V\text{-nus}\) language is the view statement. The data structure View and the view statement have their origin in the language Booster as it is described in [77]. The view is suited to specify selections of a data structure. A view is a data structure with references to existing data. The dimension of a view \(V\) on a data structure \(W\) can be less than, equal to, or greater than the dimension of \(W\).

The view statement `view ld V` assigns the view function of the view \(V\) to the view identifier \(ld\). The terms ‘view’ and ‘view function’ are often interchanged; they mean the same. However, ‘view’ is often used in a context where we refer to some data structure; ‘view function’ is used where we focus on the function that characterizes this data structure.

In Example 2.3.3 a view is created on the diagonal of a matrix.
2.3. PROGRAM CONSTRUCTS

Example 2.3.3 A view on the diagonal of a matrix.
Consider a matrix $A$ of $n \times n$ elements. A view $V$ can be constructed on the diagonal of $A$ as follows:

$$\text{view } V \left ( [(i,n)] \right ) \left ( A, \left [i,i\right ] \right )$$

The view $V$ is a one dimensional data structure of rank $n$. The term $[(i,n)]$ denotes an index set $\{0, \ldots, n-1\}$. The view function in question refers for each $i \in \{0, \ldots, n-1\}$ to the element $A_{(i,i)}$.

A view is used as a shortcut to a specific selection in a data structure. By accumulating views a complex selection can be defined that can be easily referenced. Since a view is a function from indexes to locations, a view is not dependent on other views. In Example 2.3.4 a program fragment is given where views are created in terms of other views.

Example 2.3.4 A view on a view.
Assume that data structures $A$ and $B$ are of rank $n$, and have a different content. Consider the following statement sequence:

$$\text{view } V \left ( [(i,n)] \right ) \left ( A, \left [i\right ] \right ),$$
$$\text{view } W \left ( [(i,n)] \right ) \left ( V, \left [i\right ] \right ),$$
$$\text{view } V \left ( [(i,n)] \right ) \left ( B, \left [i\right ] \right )$$

After executing these three statements, the view $V$ "views" all elements of $B$. The view $W$ views all elements of $A$; it is not altered by the last view statement.

The meaning of assignment $D \ E$ is straightforward: the value of the expression $E$ (after evaluation) is assigned to the data structure $D$. This data structure must represent exactly one location. Consequently, the expression $E$ must be a scalar. Some languages, for instance HPF [41], have array assignments that enable the programmer to specify that the whole content of an array needs to be assigned to another array. In $V$-nus this can be expressed with the Forall statement. Example 2.3.5 shows how an array assignment is expressed in $V$-nus. The notion of a selector list is used, which is explained later in this section.
Example 2.3.5 How to assign one data structure to another. Consider two arrays/data structures A and B of rank n. The array assignment (of a pseudo language): A := B is expressed in V-nus as:

\[
\text{forall } [(i, n)] \text{ statements } \text{assignment } (A, [i]) (B, [i])
\]

Next to a variable representing a data structure to which it is assigned, it is also possible to use a variable representing a function call in an assignment. This makes only sense when the function call returns a data structure of type 'view', because the result of a function call does not exist outside the statement in which the function was called. Assigning to a view has its effect in the referenced data structure. As such, a function can be used as a mechanism to assign to a temporary (complex) view on a given data structure. If a function call returns a data structure that is not of type view to which is assigned, the assigned data become garbage since there is no reference to this data when the statement is finished. This subject is treated in more detail in Section 4.5.2.

The statement procedurecall Id Ps is a call to the declared procedure Id with actual parameters Ps.

The statement LibraryEntry is present in order to use existing libraries of statements (or procedures). The list of expressions can be used to refer to a specific statement (or procedure) with its arguments. For instance, referring to a scalar initialization (sinit) of an existing library that assigns a constant c to a variable x can be expressed in V-nus with the statement

\[
\text{library } [\text{sinit}, \text{x}, \text{c}]
\]

(See also Section 7.6).

2.3.6 Expressions

An expression can exist in many different forms. Briefly, we will explain the different appearances of an expression.

Composing an expression When an operator is needed in an expression, a tuple notation is used. A unary operator op for an operand E requires a 2-tuple of the form \((op, E)\). A binary operator op for the operands \(E_1\) and \(E_2\) is used as \((E_1, op, E_2)\). As with statements, expressions can be annotated by pragmas, which precede the expression that is annotated.

A special class of the expressions consist of set expressions. Set expressions are constructed with the operators 'U', '∩', '\\', and '...' which stand for union, intersection, difference, and succession respectively. Set expressions will always be used in such a way that
the expression in which these are embedded produce a scalar as a final result. The operator
that can reduce a set expression to a scalar expression is called the *set selector*; notated
as ‘!’. A (scalar) expression of the form $(e, 1, se)$ returns the $e^{\text{th}}$ value of the set $se$, where
$se$ can be ordered with the ‘less than’ operator ‘$<$’.

**Selections** A *Selection* represents exactly one element of a data structure. Data struc-
tures in *V-nus* are ‘flat’; that means that the elements of a data structure are a scalar
(and not another multidimensional data structure). A *Selection* can be used as a data
structure (of type scalar) that has a dimension of zero. In a selection $(E, S)$ the number
of *Selectors* in the selector list $S$ must be equal to the dimension of the expression $E$,
where $E$ represents a data structure. As such it is guaranteed that exactly one element is
selected from the data structure $E$. If, for instance, $A$ is a two dimensional data structure,
the *Selection*

$$(A, [7, 13])$$

is a valid *Selection* (provided that the selectors 7 and 13 do not exceed the accompanying
ranks).

**Views** The language construct *View* is already introduced with the view statement. A
*View* is a data structure that allows for the construction of variables that only exist within
that data structure. In the *View* $(Cs, Selection)$ the cardinality list $Cs$ introduces these
local variables which can be used in the notation for the *Selection*. With a view it is
possible to denote, for instance, all elements of a data structure with an even index as
follows

$$([(i, (n, \text{DIV}, 2))], (A, [(i^*, 2)])$$

where $A$ is assumed to have rank $n$.

**Shapes** The term *shape* $[e_0, \ldots, e_{n-1}]$ denotes a data structure. The expressions $e_0$
to $e_{n-1}$ must have equal ranks. In case these expressions represent scalars this ‘shape’ has a
rank of $n$. In case the expressions are again shapes (views are not allowed), then the rank
of the presented shape is the sequence $n$ succeeded by the rank of the expressions. The
shape construction is used to specify a nameless data structure. At least one expression
must be present in the shape construct.

**Function call** A function call refers to a previously declared function, where the actual
parameters are passed by value to the formal parameters. The result of a function call is
a data structure.
Ownership  The distribution of data among the processes assigns an owner to all data. The ownership primitives can be used to see which process owns a certain data structure. The expression \textit{owner} \(D\) returns an encoded set of processes that own the data structure \(D\). When data is replicated among the processes it is possible to have more than one owner for a data element. The expression \textit{isowner} \(D\ E\) returns ‘true’ if the process indicated by the number \(E\) is owner of the data structure \(D\). Note that this expression can be used to check for ownership without encoding the specified process. Since it is possible to have multiple owners for a data structure, it makes sense to check which process actually sent the data structure (when it is sent). This can be done by the expression \textit{sender} \(D\). This expression returns the owner of \(D\) that the run-time system holds responsible for sending to other processes. If a data structure is not replicated, \textit{sender} \(D\) and \textit{owner} \(D\) return identical results.

Reduction  A common language construct in data parallel languages makes it possible to ‘reduce’ a data structure with multiple elements to one element, according to a given operator. For instance, the \textit{V-nus} expression \textit{reduce} \(+\) \(\{(i,n)\}, A, [i]\) denotes such an expression. It represents the sum of all elements \(A_i\), for \(0 \leq i < n\). Of course, the same result can easily be computed by denoting an \textit{Iteration}. But having a single language construct expressing reductions gives the compiler the opportunity to generate a much more efficient computation strategy, that may involve parallelism. More on this subject can be found in Section 5.8.

2.4 Defining the semantics

In this thesis we will not formally define the meaning of the whole language \textit{V-nus}. The constructs we are interested in have been presented in the preceding section. The other constructs are present in the language for making it a ‘real’ language, i.e. a language that can be truly used on its own. For instance, \textit{Action} can be a \textit{Print} action of the form \textit{print} \(P\). Such an action is meant to perform I/O statements. Without doubt it is necessary to have this expressiveness for a self contained language, but this thesis will not incorporate it into the formal semantics.

In specifying the formal semantics of the presented \textit{V-nus} constructs, we will assume that the \textit{V-nus} fragments are correct: i.e. no syntax errors, type check errors, or other errors that can be detected at compile time are present in these \textit{V-nus} fragments. We say that only ‘valid’ \textit{V-nus} programs are considered. It is out of the scope of this thesis to describe the mechanisms needed to check for these errors.
Chapter 3

The semantic states

At the root of every claim we make to possess an understanding of the physical world sits a mathematical truism. But upon what does this faith in mathematics rest? How can our inky squiggles on pieces of paper possibly tell us how the world goes round? Why is the world found to be so unerringly mathematical?

*Pi in the Sky*

John D. Barrow

3.1 Introduction

In constructing the denotational semantics for the language *V-nus*, we make use of some basic mathematical concepts. First, we will give a flavor of the ideas behind the concepts to be defined later.

A semantic model for *V-nus* is a construction that can be used to compute the meaning of *V-nus* programs. However, 'the meaning' of a program is not a concrete datum; it depends on the interest one has in a program. For instance, one can only be interested in the combination 'input - output' of a program, but concerns about intermediate results can make out a case as well. The semantic model used in this thesis will reflect the activities of a program in a mathematical object: the program state. For the construction of the program state we went along the following lines. Mainly, the value of the program variables must be recorded. These program variables may represent scalars and arrays, where scalars are, in fact, arrays with an empty index space. Instead of making a relation between a program variable and its data, we relate a program variable to memory locations. Another relation is used to link memory locations to data. We use functions to define these relations.

These functions are concrete items that are present in the program state. 'The meaning' of a program is then presented in the form of this program state. It depends on the way this program state can be handled whether we can compute the relation of input and output of a program only, or that it is possible to compute the program state for program parts as
well. This subject is dealt with in Chapter 4. In this chapter we focus on the construction of the signature of a program state. We show some examples how this program state can be used. However, the formal access functions on this program state are defined later in this thesis.

The main data object in V-nas is the 'view'. A view is a function from an index set to the set of locations (see Figure 3.1). Such an index set is called a bounded set, which means that it has a lower and an upper bound. The set of all bounded sets is defined; the function representing a view is defined as something that is called an index propagation function.

The set of locations represents the memory of a machine. In case we deal with a shared memory system, the set of locations is a representative for all memory present in the machine. For distributed memory systems, the set of locations represents the memory of a process. This case is explained in more detail in Chapter 6.

Figure 3.1: The views are declared in terms of shapes, but are defined by ordinary index propagation functions. \( f, g, h, k \) and \( l \) are index propagation functions. The set of locations is a bounded set, each location representing a memory address.

Execution of V-nas code is reflected by a change of the program state. In order to change the functions in the program state, a replacement function is defined that can adjust a function definition. For instance, an assignment to a variable \( A \) implies a change for the function representing this variable \( A \), which resides in the program state.

In Section 3.2 some basic concepts are introduced. Thereafter, we use these concepts to construct the program state in Section 3.3. In Section 3.4 we pay special attention to the meaning of indexes in our semantic model, and define an index state. Section 3.5 introduces a way to cope with declarations and items of the symbol table. At the end of this chapter, in Section 3.6, we define some functions for using the constructed program state. Only then, we make an explicit relation between an index space and its accompanying data structure.
3.2. TOWARDS A PROGRAM STATE

3.2.1 Notational conventions

This chapter defines several functions for which we use the following notational conventions:

- Functions representing the program state, its substates, or the index state are denoted with a Greek character.
- Functions that are used as an operator are represented by a special symbol; i.e. a character that is not present in the English alphabet.
- The notation $\mathbb{P}(X)$ is used to denote the power set of $X$.
- For a Selection $(\mathcal{A}, [\text{exprs}])$ in V-nus, we denote its mathematical equivalent as $\mathcal{A}_{[\text{exprs}]}$, where exprs and $\tilde{\text{exprs}}$ represent V-nus expressions and corresponding mathematical expressions respectively. For instance, the value of the V-nus Selection $(\mathcal{A}, [(3, \cdot, 5), 13])$ can be denoted by the mathematical expression $\mathcal{A}_{(3\cdot, 13)}$ (The tilde (\tilde{\cdot}) will be omitted, if it is clear what kind of expression is meant.)

3.2 Towards a program state

The program state we will use for the semantic models is a tuple of functions. Since we heavily rely on these functions to express the semantics of program constructs, we will first define what we mean by a function.

**Definition 3.2.1** A partial function $f$ from a set $X$ to a set $Y$ is a relation $f \subseteq X \times Y$ such that

$$\forall x \in X \forall y, y' \in Y: ((x, y) \in f \land (x, y') \in f) \Rightarrow y = y'$$

 Defined.

We will use the notation $f(x) = y$ when there is a $y$ such that $(x, y) \in f$. We say that $f(x)$ is defined. We will write $f(x) = \perp$ if there is no $y$ such that $(x, y) \in f$, and we say that $f(x)$ is undefined. We extend the definition of a partial function in order to define a total function.

**Definition 3.2.2** A total function $f$ from a set $X$ to a set $Y$ is a partial function from $X$ to $Y$ such that

$$\forall x \in X. \exists y \in Y. f(x) = y$$

 Defined.

For a function $f \subseteq X \times Y$ we will also write $f : X \rightarrow Y$ and say that $f$ is of type $X \rightarrow Y$. If we do not explicitly call a function a ‘total function’ we assume it to be a ‘partial function’.

We continue with the introduction of different sets that are used in the program state functions. Typically, a semantic function that resides in the program state is of type $I \rightarrow F$, where $I$ is a set of identifiers and $F$ is a function representing the meaning of the views identified by $I$. The following sets of identifiers are distinguished.
**Definition 3.2.3** The set of identifiers \( I_D S \) which can be used in a \( V-nus \) program is partitioned into the following subsets.

- \( VI \) is the set of *view identifiers*
- \( II \) is the set of *index identifiers*

The different sets of identifiers are labeled such that \( VI \cap II = \emptyset \).

The view identifiers will represent all data structures that are used in a program, including scalars. A data structure is seen as a view on a selected set of locations which, in turn, represent data (see Figure 3.2). This will be clarified later in this chapter. The index identifiers will represent the indexes of the data structures. Example 3.2.1 gives a \( V-nus \) view statement in which a view identifier and an index identifier occur. So, a view identifier represents the relation between an index set and its locations. An index identifier simply represents an index value. Typically, this index identifier can be input to the function represented by the view identifier.

**Example 3.2.1 Distinguishing identifiers.**
The view statement:

\[
\text{view } V \left( ([i,4]), (A, [i]) \right)
\]

uses the index identifier \( i \in II \) and the view identifiers \( V, A \in VI \).

In the program state, a data structure will be represented by a function from an index set to a set of locations. Such an index set must be a finite, 'rectangular' set of vectors, called a *bounded set*. 
3.2. TOWARDS A PROGRAM STATE

Definition 3.2.4 Let \( l, u \in \mathbb{N}_d \), for an arbitrary \( d \in \mathbb{N} \); say \( l = (l_0, \ldots, l_d) \) and \( u = (u_0, \ldots, u_d) \). A bounded set \( \mathbb{N}^{(l,u)}_d \) is defined as

\[
\mathbb{N}^{(l,u)}_d = \{ x = (x_0, \ldots, x_d) | l_i \leq x_i \leq u_i, \ 0 \leq i \leq d \}
\]

The set \( BSS \) consisting of all possible bounded sets is defined as

\[
BSS = \{ \mathbb{N}^{(a,b)}_c | a, b \in \mathbb{N}, \ c \in \mathbb{N} \}
\]

The notation \( \mathbb{N}^{(a,b)}_c \) is used as a shortcut for \( \mathbb{N}^{(a,b)}_c \), where \( c \) is the rank of \( a \) and \( b \).

Bounded sets consist of vectors, which are notated as tuples. That is, two or more elements that make a vector are placed between brackets and are separated by commas. One element has no need for a pair of brackets, such that we do not distinguish the notations \((n)\) and \(n\), where \(n\) is a single element. For all sets of tuples we say that the empty tuple () is called the 'undefined value'. Note that the empty tuple is not a representation for \(\bot\).

For instance, if we have \( f(x) = () \) then \( f \) is defined for \( x \), and thus \( f(x) \neq \bot \).

A zero-based bounded set \( \mathbb{N}^{(l,u)}_d \) is a bounded set for which we have: \( l \in \{0\}^d \).

In order to be able to select an element of a vector we need projection functions. We define the generic functions \( \text{proj}_i \) that select the \( i \)th element of a vector. Note that we count from zero, which makes the 'first' element in fact the 'zeroth' element.

Definition 3.2.5 Let \( X \) be an arbitrary set of vectors of length \( n+1 \), where the elements of the vectors are of type \( N \). The projection functions \( \text{proj}_i : X \to N \) are defined as

\[
\text{proj}_i((x_0, \ldots, x_i, \ldots, x_n)) =
\begin{align*}
& x_i & \text{if } 0 \leq i \leq n \\
& \bot & \text{otherwise}
\end{align*}
\]

We assume the existence of a special bounded set, called the set of locations, which is an abstract representation of the memory locations. This set is referred to as \( \text{Loc} \). The data stored in these memory locations are elements of the set \( \text{Data}_0 = \mathbb{N} \cup \mathbb{R} \cup ... \). Only elements of \( \text{Data}_0 \) can be stored in memory locations. For the purpose of handling functions using truth values we collect these values in the set \( \mathbb{B} \), such that \( \mathbb{B} = \{\text{false}, \text{true}\} \). We will insist that \( \mathbb{B} \subseteq \text{Data}_0 \). The following definition formalizes these mentioned sets.

Definition 3.2.6 The set \( \text{Loc} \) of locations is a special bounded set for which holds

\[
\text{Loc} \in BSS \cup \{\ell\}
\]

where \( \ell \) is a special element. The set \( \text{Data}_0 \) of primary data is a set of 'numbers', specified by

\[
\text{Data}_0 = \mathbb{N} \cup \mathbb{R} \cup ...
\]

This set is extended to a set \( \text{Data} \) in which tuples are included as well. The set \( \text{Data} \), dependent on the set \( \text{Data}_0 \), is recursively defined as follows.
1. $Data_0 \subseteq Data$

2. if $x_0, \ldots, x_n \in Data$ for $n \in \mathbb{N}$ then $(x_0, \ldots, x_n) \in Data$

3. $Data$ is the smallest set such that 1 and 2 hold.

The set $\mathbb{B} \subseteq Data_0$ of truth values is denoted as

$$\mathbb{B} = \{false, true\}$$

### 3.3 The construction of the program state

Here we introduce the notion of program state that is used in the semantic model of $V-nus$. The program state is a mathematical object (here: a tuple) that holds the information of the state of a $V-nus$ program. Its type is denoted as $\Sigma$. The program state that is defined in this section consists of two parts, called substates: one recording the state of the view identifiers, and one recording the state of the memory locations.

Information about the index identifiers is kept outside the program state, and is recorded in a separate index state. The reason for this is that indexes play an important role in using data structures in $V-nus$. Therefore, it is convenient to have a separate mathematical object to investigate and manipulate, without influencing the program state.\(^1\)

All information that is given in the form of 'declarations' in a $V-nus$ program, is also stored in a separate state, called the declaration state. This declaration state will also represent information that is stored in the symbol table. High level language declarations are represented by entries in the $V-nus$ symbol table, and, consequently, reflected in the declaration state.

#### 3.3.1 Scopes and locations

As mentioned in Section 2.3 $V-nus$ functions are referentially transparent. Data access in a function body can only use the function arguments and locally declared variables. Note that variables only represent bounded sets of locations (see Section 2.2): function and procedure declarations 'live' in the symbol table and are visible in the entire program. In this thesis we assume that data access in procedures also uses its arguments and locally declared variables only. Note that this has no impact on the expressiveness of these procedures with respect to procedures that can access global variables. 'Global variables' (when these are present in a high level language) can always be passed as a parameter to functions and procedures.

---

\(^1\)The term 'program state' may be confusing since indexes do influence the state of a program. However, our 'program state' closely resembles the common idea of a program state as it is used in the literature on semantics, which justifies the term.
3.3. THE CONSTRUCTION OF THE PROGRAM STATE

We use the notion of a scope of a program construct to identify the set of identifiers that are accessible in that program construct. The scoping mechanism we use in V-nus is simple. For each program construct only the identifiers within the current scope are accessible. Scopes are outlined by a function body, a procedure body, or the entire program. The scoping mechanism works like a stack. Entering a new scope is like pushing a set of visible variables on the stack. Leaving a scope pops the top-most set off the stack.

The representation of a scope as it is used in our models requires a set of unique elements on which a linear order on all elements is possible. In fact, a scope value \( n \) means that the current scope (read: the set on top of the scope stack) is the \( n^{th} \) set on the stack. In the remainder of this thesis we use the term ‘scope’ for the set of visible variables as well as for the value representing this set. It will be clear from the context what is meant.

**Definition 3.3.1** The set \( \text{Scope} \) is defined as

\[
\text{Scope} = \mathbb{N}
\]

We will use a pair \((Id, s)\) as a reference to the identifier \(Id\) in the scope \(s\).

The location state represents the memory locations and the values these contain. The location state is a substate of the program state and is defined as follows.

**Definition 3.3.2** The set \( \Sigma_L \) of location state functions is defined as

\[
\Sigma_L = \{ \sigma_L : \text{Loc} \to \text{Data}_0 \}
\]

For the special element \( \ell \in \text{Loc} \) we say that \( \sigma_L(\ell) \) has the current scope as result.

### 3.3.2 The view state

The view state links the indexes of a certain data structure to the accompanying locations. These links are constructed with index propagation functions, already known as ‘view functions’. Since the indexes and the locations are represented by bounded sets, the index propagation functions have a bounded set for the domain and codomain.

**Definition 3.3.3** The set \( IPS \) of index propagation functions (ip functions for short) is defined as

\[
IPS = \bigcup_{bs_1, bs_2 \in USS} \{ ip : bs_1 \to bs_2 \}
\]

A view state function makes it possible to assign an index propagation function to an identifier in a certain scope.
**Definition 3.3.4** The set \( \Sigma_V \) of view state functions is defined as

\[
\Sigma_V = \{ \sigma_V : (VI \times Scope) \rightarrow IPS \}
\]

\[\square\]

Example 3.3.1 sketches how an assignment in \( V-nus \) is reflected in a program state.

**Example 3.3.1** *Recording a view in the program state.*

Consider the following \( V-nus \) term where \( A \) is a one-dimensional data structure. The assignment sets the second element of \( A \) to 13.

\[
\text{assignment} \ (A, \ [1]) \ 13
\]

Let's call \( s \) the current scope. This will result in some program state \( \sigma \) such that for the substate \( \sigma_V \) we have that \( \sigma_V(A,s)(1) = l_1 \), for some location \( l_1 \), and for the substate \( \sigma_L \) we have \( \sigma_L(l_1) = 13 \).

---

### 3.3.3 The program state

With the definition of the view state \( \Sigma_V \) and the location state \( \Sigma_L \) we can present how the program state for \( V-nus \) programs is constructed.

**Definition 3.3.5** The set of program states \( \Sigma \) (with typical elements \( \sigma, \sigma', \ldots \)) is defined as

\[
\Sigma = \Sigma_V \times \Sigma_L
\]

\[\square\]

In presenting the semantics of \( V-nus \) we will use some convenient abbreviations. Example 3.3.2 shows the utilization of the following notation. If a state \( \sigma \in \Sigma \) exists we also assume the existence of the substates \( \sigma_V \) and \( \sigma_L \) such that \( \sigma = (\sigma_V, \sigma_L) \). For the different
dimensions of a program state $\sigma \in \Sigma$ we use the following notations for an arbitrary operator $\otimes$: $\sigma \otimes_x X = \sigma'$ such that $\sigma'_x = \sigma_x \otimes X$ and for $y \neq x$: $\sigma'_y = \sigma_y$ where $x, y$ stand for one of the subscripts $V, L$. For two program states $\sigma, \sigma' \in \Sigma$ we write $\sigma \otimes_\cdot \sigma'$ as a shortcut for $\sigma \otimes_V \sigma'_V \otimes_L \sigma'_L$, assuming that $\otimes$ is left associative.

Example 3.3.2 Using an operator on program states.
Consider an arbitrary program state $\sigma = (\sigma_V, \sigma_L)$. Let’s assume that the substate $\sigma_V$ needs to be extended, where the extension is defined in a set $X$. So, $\sigma_V$ and $X$ have the same type. A new program state $\sigma'$ in which the extension is recorded, can be expressed as $\sigma' = \sigma \cup_V X$ which is defined as $\sigma' = (\sigma_V \cup X, \sigma_L)$.

Remark: ‘updating’ a program state as is presented in Example 3.3.2 is formalized in Section 3.6.

Mathematical intermezzo
As in most of the literature on denotational semantics we too will introduce the concepts of a partially ordered set (po), a complete partially ordered set (cpo), a least upper bound (lub), and a chain. These concepts are needed when we have to deal with loops. It is trivial to write down a loop in V-nus that never stops (for instance: while true Statements). In Section 4.7 special attention is paid to semantic functions that describe such a loop (such that these functions are well-defined), where the above mentioned concepts are used.

We follow de Bakker [6] in defining these concepts. We will end this intermezzo with the assertion that the set of program states $\Sigma$ is a cpo.

Definition 3.3.6 Let $X$ be an arbitrary set. The relation $\sqsubseteq \subseteq X \times X$ is called a partial order when it is a reflexive, antisymmetric, transitive order. We say: $(X, \sqsubseteq)$ is a partial ordered set (po for short).

In a partially ordered set a notion of a least upper bound exists. The least upper bound of a set $X$ is described in words as the ‘smallest’ element that is ‘greater’ than (or equal to) all elements of $X$.

Definition 3.3.7 Let $(Y, \sqsubseteq)$ be a po, and let $X \subseteq Y$. $y \in Y$ is called the least upper bound of $X$ (notation: lub $X$) if

1. for all $x \in X$ it applies that $x \sqsubseteq y$

2. for all $y' \in Y$ and all $x \in X$, if $x \sqsubseteq y'$ then $y \sqsubseteq y'$.
In order to define a complete partially ordered set we need the notion of a chain. A chain is an ascending (or descending) sequence of elements of a po with respect to its partial order.

**Definition 3.3.8** An ascending chain on a po \((X, \sqsubseteq)\) is a sequence \(x_0, x_1, \ldots\) such that for all \(i \in \{0, 1, \ldots\}\) : \(x_i \sqsubseteq x_{i+1}\).

A complete partially ordered set is now defined as follows.

**Definition 3.3.9** A complete partially ordered set \((X, \sqsubseteq)\) (cpo for short) is a po \((X, \sqsubseteq)\) which satisfies the following requirements:

1. There is a least element \(\bot\) such that for all \(x \in X\) : \(\bot \sqsubseteq x\)
2. Each chain \(x_0, x_1, \ldots\) with \(x_i \in X\) for \(i \in \{0, 1, \ldots\}\) has a lub (notation: \(\text{lub}_{i \in \{0, 1, \ldots\}} x_i\)).

Now, an example of a partial order \(\sqsubseteq\) on program states is given with which \((\Sigma, \sqsubseteq)\) is a cpo. For all \(\sigma, \sigma' \in \Sigma\) we define:

\[
\sigma \sqsubseteq \sigma' \iff (\sigma = \bot \text{ or } \sigma = \sigma')
\]

It is trivial to show that now \((\Sigma, \sqsubseteq)\) is a cpo (a so-called discrete cpo).

### 3.4 Using indexes

Separate from the program state we will use an index state. This state will represent the current values of the index identifiers. Since index identifiers need no preceding declaration, it is stated that their scope is the 'duration' of the statement in which they are used. This requires a different scoping mechanism than is used for the other identifiers. Furthermore, accessing data by means of indexes is an important mechanism in V-nus. By handling these in a separate entity, we hope to present a clearer view on the meaning of these indexes.

We will have a program state that steadily 'grows' when the program evolves. More precisely, it means that the number of elements in the domains of the substates increases (or remains the same) when using access functions on a program state. It is not possible to delete something from the program state; i.e. decrease the number of elements in a substate domain. On the other hand, the index state can be seen as a fluctuating, temporary storage. It is possible to add elements to and delete elements from an index state. We think it is clear-cut to separate the program state and the index state.

**Definition 3.4.1** The set \(\Phi\) of index state functions is defined as

\[
\Phi = \bigcup_{bs \in BSS} \{ \varphi \mid \varphi : II \to bs\}
\]
This definition specifies more index state functions than we will use. Indexes are to be of type 'bounded set' which consist of 'one-tuples', i.e. tuples that consist of exactly one value. In Example 3.4.1 the utilization of index state functions is shown.

Example 3.4.1 Mapping index identifiers to their values.
Consider a one dimensional data structure $A$. The next $\text{V-nus}$ term creates a view $V$ on the data structure $A$ such that only the first two elements of $A$ are viewed.

\[
\text{view } V \left(\{[i,2]\}, \{A,[i]\}\right)
\]

During the evaluation of this view statement the index identifier $i$ will have the values 0 and 1. Therefore, two index state functions $\varphi_1$ and $\varphi_2$ are created such that: $\varphi_1(i) = 0$ and $\varphi_2(i) = 1$.

3.5 The declaration state

The symbol table is present in every complete $\text{V-nus}$ program and represents all kinds of information about the program’s variables. While determining the meaning of a $\text{V-nus}$ program it is sometimes necessary to retrieve information from this symbol table. For instance, to present correct semantics of a data structure of type $\text{shape}$, the rank of this data structure must be known. This information is stored in the symbol table $\text{SymbolTable}$. In our semantics we will use an abstract representation of the $\text{SymbolTable}$, denoted as $\delta \in \Delta$, and is called the declaration state.

A $\text{V-nus}$ program is always of the form: program $\text{Initializations} \ Declarions \ Statements \ \text{SymbolTable}$. The declaration state will represent the initializations, the declarations, and the information of the symbol table as given in the appropriate program parts. It is out of the scope of this thesis to define the declaration state. It will do to describe the declaration state as is done below.

Because $\text{V-nus}$ is an intermediate language some declarations are represented in the $\text{SymbolTable}$, and some are represented in the $\text{Declarations}$. Consider a language $L$ that can be translated to $\text{V-nus}$. Simple declarations (i.e. declarations that are no function- or procedure declarations) in $L$ must be translated to certain $\text{Entries}$ in the $\text{SymbolTable}$. Function- and procedure declarations in $L$ need to be translated to the appropriate $\text{V-nus}$ declarations in $\text{Declarations}$.

Definition 3.5.1 The set $\Delta$ of declaration state functions is defined as

\[
\Delta = \{\delta \mid \delta : (IDS \times Scope \times Constructors) \rightarrow \text{Vnus}\}
\]

where $\text{Constructors}$ is a set of keywords that can be used to retrieve specific information of a certain identifier.

\[\square\]
The set *Constructors* will not be formally defined. Each time an element of *Constructors* is used in accessing the declaration state, it is explained on the spot. In Example 3.5.1 a few examples are given of using *Constructors* in the declaration state functions.

**Example 3.5.1** Using *Constructors* in the declaration state.
Consider the following function declaration in V-nus:

```plaintext
function F [x1, x2] y statements [
    assignment (y, []) ((x1, []), +, (x2, []))
]
```

Assume that the scope for this function declaration is \( s \). A declaration state \( \delta \) will represent this function declaration such that:

\[
\delta(F, s, \text{stmt}) = \text{statements} \left[ \text{assignment} \ (y, []) \ ((x1, []), +, (x2, [])) \right]
\]

\[
\delta(F, s, \text{par}) = [x1, x2]
\]

\[
\delta(F, s, \text{result}) = y
\]

### 3.6 Access and modification functions

Special functions and notations are needed to use the program state and the index state in a semantic model. A frequently used function is the *replacement function* that, among others, is used to alter the substates of the program state.

**Definition 3.6.1** Let \( f, g : X \rightarrow Y \) be functions on arbitrary sets \( X \) and \( Y \). Let \( Z = \mathcal{P}(X \times Y) \). The replacement function \( \triangleleft : Z \times Z \rightarrow Z \) is defined such that

\[
(f \triangleleft g)(x) = \begin{cases} 
  f(x) & \text{if } g(x) = \bot \\
  g(x) & \text{otherwise}
\end{cases}
\]

where \( \triangleleft \) is left-associative. \( \Box \)

Sometimes the shorthand notation \( f \triangleleft (x, y) \) is used to express \( f \triangleleft \{(x, y)\} \). See Example 3.6.1 for an alteration of an existing function.
Example 3.6.1 Changing a function by a replacement.
Let \( f : \mathbb{N} \to \mathbb{N} \) be the identity function (so \( f = \{(0,0),(1,1),(2,2)\ldots\} \)). The result of \( f(1) \) can be changed to 13 with the replacement \( f \circ (1,13) \). We have that \( f \circ (1,13) = \{(0,0),(1,13),(2,2)\ldots\} \).

The next function we introduce will concatenate two index vectors. Concatenation is needed when nested loops are considered. Within a loop nest an index space can be used that is the concatenation of the index spaces of the outer loops. This is how one value can be addressed out of a multi dimensional data structure. Example 3.6.2 gives a flavour of the use of this concatenation function in relation with loop nests. With the next definitions we will define the relation between an index set and a data structure.

Definition 3.6.2 Let \( bss = \bigcup_{bs \in \text{S}} bs \) and \( B = \text{S} \cup bss \). Let \( x = (x_0, \ldots, x_n) \) and \( y = (y_0, \ldots, y_m) \) be arbitrary vectors such that \( x, y \in bss \). Let \( bs, bs' \in \text{S} \) be arbitrary bounded sets. The concatenation function \( \bullet : B \times B \to B \) is defined as

\[
(x_0, \ldots, x_n) \bullet (y_0, \ldots, y_m) = (x_0, \ldots, x_n, y_0, \ldots, y_m)
\]

\[
x \bullet bs = \begin{cases} x & \text{if } bs = \emptyset \\ \{x \bullet b | b \in bs\} & \text{otherwise} \end{cases}
\]

\[
bs \bullet x = \begin{cases} x & \text{if } bs = \emptyset \\ \{b \bullet x | b \in bs\} & \text{otherwise} \end{cases}
\]

\[
bs \bullet bs' = \bigcup_{b \in bs} b \bullet bs'
\]

For a tuple \( x = (x_0, \ldots, x_n) \) we say that \( x_i \) is an element of \( x \) for \( 0 \leq i \leq n \) and \( n > 0 \). So, a one-tuple is not an element of itself. All elements of \( x_i \) are also elements of \( x \). \( x_i \) is called a base element of \( x \) iff \( x_i \) is an element of \( x \) and \( x_i \) has no elements.
Example 3.6.2 The concatenation function related to loop nests.
Consider the following V-nus program:

```
iteration [(i1, n1), (i2, n2)] statements [
    iteration [(j, m)] statements [
        assignment (A, [i1, i2, j]) (B, [13, i1, i2, j])
    ]
]
```

The index set of the outer loop can be represented by the bounded set: \( b_{s_1} = \mathbb{N}_{0}^{(0,0),(n1-1,n2-1)} \). For the inner loop we use the bounded set: \( b_{s_2} = \mathbb{N}_{0}^{(0,m-1)} \). The index space for which the assignments must be performed is constructed by a concatenation of the bounded sets \( b_{s_1} \) and \( b_{s_2} \). This new index set is expressed as a bounded set \( b_{s_3} = b_{s_1} \cdots b_{s_2} \). Accessing elements of the data structure \( B \) requires concatenation of a scalar \( (13) \) with a vector \( ([i1, i2, j]) \in b_{s_3} \). To address these elements of \( B \) we use the expression: \( 13 \cdot (i1, i2, j) \) giving us the vector \( (13, i1, i2, j) \) to refer to \( B_{(13,i1,i2,j)} \).

We will now introduce the notion of a spanning index set for a data structure. A spanning index set \( is \) for a data element \( d \) is a set of tuples over \( \mathbb{N} \) such that each base element of \( d \) can be referred to by a unique index \( i \in is \) and there is no index \( i' \in is \) that does not refer to a base element of \( d \). The function that returns a spanning index set for a given data structure is called \( \text{Span} \) and uses the vector-set concatenation of Definition 3.6.2. First, a function \( \text{Span}' \) is defined. Then, \( \text{Span} \) is defined in terms of \( \text{Span}' \) such that the empty tuple (the 'undefined value') can be used properly.

**Definition 3.6.3** Let \( d_0, \ldots, d_n \in \text{Data} \) for \( n \in \mathbb{N} \). Let \( b_{s} = \bigcup_{bs} bs \). The function \( \text{Span}' : \text{Data} \rightarrow \mathcal{P}(bs) \) is defined as

\[
\text{Span}'((d_0, \ldots, d_n)) = \begin{cases} 
\emptyset & \text{if } n = 0 \text{ and } d_0 \in \text{Data}_0 \\
\{i \cdot \text{Span}'(d_i) | 0 \leq i \leq n\} & \text{otherwise}
\end{cases}
\]

It is required that \( () \notin \text{Span}'(d) \). Now let \( d \in \text{Data} \) be an arbitrary valid argument for \( \text{Span}' \). The function \( \text{Span} : \text{Data} \rightarrow \mathcal{P}(bs) \) is then defined as

\[
\text{Span}(d) = \begin{cases} 
\{()\} & \text{if } \text{Span}'(d) = \emptyset \\
\text{Span}'(d) & \text{otherwise}
\end{cases}
\]

For referring to certain base elements of a data structure we introduce a special notation.
3.6. ACCESS AND MODIFICATION FUNCTIONS

Notation: let \( d \in \text{Data} \) and let \( i \in \text{Span}(d) \); say \( i = (i_0, \ldots, i_n) \). Then \( d_i = \text{proj}_{i_n}(\cdots \text{proj}_{i_1}(d) \cdots) \). Note that \( d_{(1)} = d \). We say that \( i = \text{Span}(d) \) is a spanning bounded set iff \( i \in BSS \).

Example 3.6.3 A spanning bounded set.
Consider the data element \( d = ((a, b, c), (d, e, f)) \). The spanning bounded set for \( d \) is:
\[
\text{Span}(d) = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)\}.
\]
As such, we have, for instance:
\[
d_{(1,2)} = \text{proj}_2(\text{proj}_1(d)) = \text{proj}_2((d, e, f)) = f.
\]

We end this section by defining the relation between an index set and the data that is referenced. So, we have a bounded set that represents an index set of a data structure. An index propagation function (\( ip \) function) assigns a location to each index. Such a location represents a data value (a value of \( Data_0 \)). In order to deduce the complete data structure that is referenced by an index set the function \( ip2data \) is defined. This function uses an \( ip \) function to link a bounded set to its locations, and uses the location state \( \sigma_L \) of the program state to link these locations with their data.

Definition 3.6.4 Let \( ip : bs \to \text{Loc} \) be an \( ip \) function for an arbitrary bounded set \( bs \in BSS \). The function \( ip2data : IFS \to \Sigma \to Data_0 \) is defined as
\[
\text{ip2data}(ip)(\sigma) = d
\]
where \( \forall i \in bs. \ d_i = \sigma_L(ip(i)) \)

\( \square \)

The function \( ip2data \) deduces the data element for a given \( ip \) function. It is also possible to deduce an \( ip \) function for a given data element and the locations in which it is stored. In general, more than one \( ip \) function can be constructed that correctly represents this data structure. The function \( data2ip \) defines exactly one \( ip \) function, which is arbitrarily chosen. This is done by making use of the relation \( <_a \) of Definition 4.7.3. This denotes a lexicographical order on elements of bounded sets. It is used here only to indicate exactly one index.

The definition below is quite bold, for it needs to express exactly one \( ip \) function. It all comes down to having an \( ip \) function of pairs \((i, l)\) for which it holds: \( d_i = \sigma_L(l) \), where \( d \) is the data. In case there are two locations \( l \) and \( l' \) representing the value \( d_i \), only one needs to be chosen (arbitrarily). Example 3.6.4 presents a situation where the data can be represented by two different \( ip \) functions.

Definition 3.6.5 Let \( d \in \text{Data} \) such that \( \text{Span}(d) = \{i_0, \ldots, i_n\} \) is a spanning bounded set where \( i_0 <_a \cdots <_a i_n \). Let \( bs = \{i_0, \ldots, i_n\} \in BSS \). Let \( \sigma \in \Sigma \). Let \( P \) be the set of permutations of \((0, \ldots, n)\) such that \( \forall p = (p_0, \ldots, p_n) \in P. \ d_{i_k} = \sigma_L(l_{p_k}) \) for \( 0 \leq k \leq n \).
Let \( q = (q_0, \ldots, q_n) \in P \) be the 'smallest' permutation of \( P \); i.e. \( \forall p \in P, q \leq_t p \). The function \( data2ip : (Data \times BSS) \rightarrow \Sigma \rightarrow IPS \) is then defined as

\[
data2ip(d, bs)(\sigma) = \begin{cases} \{(i_k, l_{q_k}) | 0 \leq k \leq n\} & \text{if } P \neq \emptyset \\ \bot & \text{otherwise} \end{cases}
\]

The function \( data2ip \) will be used for the first time in the section on the semantics of parameter passing (Section 4.5.1). It is already presented here as the counterpart of the function \( ip2data \), which both describe the relation between data elements, their index sets, and their locations.

**Example 3.6.4** Choosing one ip function for a data structure.
Assume \( d \) is the data structure \((3,3)\). The spanning bounded set is given by: \( \text{Span}(d) = \{0,1\} \). Say, \( d \) is stored in the locations \( l_0 \) and \( l_1 \). Then two ip functions may represent \( d \): \( ip = \{(0, l_0), (1, l_1)\} \) and \( ip' = \{(0, l_1), (1, l_0)\} \). The definition of \( data2ip \) is such that: \( data2ip((3,3), \{l_0, l_1\}) = ip \).
Chapter 4

The denotational semantics

"I only took the regular course".
"What was that?" inquired Alice.
"Reeling and Writhing, of course, to begin with,"
the Mock Turtle replied; "and then the different
branches of Arithmetic – Ambition, Distraction,
Uglification, and Derision".

Alice's Adventures in Wonderland
Lewis Carroll

4.1 Introduction

The denotational semantics described in this chapter are meant to 'denote' the meaning of V-nus programs. These semantics are compositional such that the meaning of parts of a program can be derived without a need for the complete V-nus program. The functions defining these semantics are denoted like functions of a functional language in the sense that these are defined in a constructive way. The semantic functions are therefore 'almost' executable in a functional language. For some functions an existential quantor is used, which is why we used the word 'almost' above. On the other hand, the element, whose existence is assumed, is taken from a finite set such that the presentation of our semantic functions is very close to an executable form.

Semantics of programming languages have been described manifoldly [6, 7, 8, 46, 47, 72, 88, 98]. However, the described programming languages are usually small, or only a part of the language is described. In this chapter we make a start with the semantic description of the V-nus language. It is a start because we leave out some language constructs that may invoke parallelism. The concept of parallelism is described later in this thesis. In Chapter 3 the different 'states' (declaration state, index state, program state) have been introduced that will be used to describe the semantics. Here, a collection of semantic functions is presented based on these states. These semantic functions are the description for the meaning of the V-nus language.

The main meaning function \( \mathcal{M} \) is used when one starts computing the meaning of a
program. The type of this function is: \textbf{Vnus} \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma. That is, when a program is given to $\mathcal{M}$ together with its declarations ($\Delta$) and its index space ($\Phi$), we are left with a relation between the ‘input state’ (the context in which the program starts) and the ‘output state’ (the program state after the program has finished). This is sufficient in case we are interested in the changes to the program state caused by the program. We can start with arbitrary valid states. In case we are interested in the meaning of the complete program, it is desirable to end up with ‘the final program state’. This requires initial states; the very first states in which the program starts executing.

The declaration state of our semantic model is not really a common object in denotational semantics. In fact, it is a representation of a table that combines pieces of \textit{V-nus} syntax. The way this declaration state is used is an alternative for the more familiar ‘substitution’ of program constructs. An often used notation is: $P[v/x]$ which means: replace all occurrences of $x$ in $P$ with $v$. The need for modifying the syntax of program constructs in denotational semantics lies in parameter passing.

De Bakker [6] uses a technique in his semantic model which he calls ‘syntactic application’. This technique is used to replace the construct $P(x, y)$ by a new piece of program text $Q(x, y)$. De Bakker: “... our way of defining the meaning of a call $P(x, y)$ may be frowned upon by purists in denotational semantics ...”. $P$ and $Q$ represent procedures, where $P$ is a procedure description in terms of formal parameter names and $Q$ is the same procedure description with the formal parameter names replaced by the actual parameter names. He assumes to have the procedure description at hand when the procedure call appears in the program. The difference with our technique is that we explicitly look up in the declaration state what the description is of the called procedure.

The technique of parameter passing used by Nielson and Nielson [72] conforms more to the general principles of denotational semantics. A procedure environment is used in which a procedure $P$ is represented by some function. At the time $P$ is called, the ‘normal’ (mathematical) function application can be used to compute the meaning of $P$ applied to its actual parameters. The semantic model used is similar to ours in that it uses a mapping from identifiers to locations, and a mapping from locations to values. At the time the procedure $P$ is declared, the formal parameters occupy locations. Passing parameters when the procedure is called is done by putting the values of the actual parameters into the locations of the formal parameters. In this way, ‘static scoping’ is used for procedures; i.e. non local variables in the procedure refer to variables in the scope of their declaration, not in the scope of the called procedure (dynamic scoping).

The same parameter passing technique is used by Slonneger and Kurtz [88]. They note that the technique leads to a ‘call-by-value’ semantics for the parameters. That means that first the value of the actual parameter is computed before the procedure is called. Winskel [98] presents techniques for both the ‘call-by-value’ and the ‘call-by-name’ semantics. In the latter, the value of the actual parameter is only computed when it is actually used in the procedure. It is not the value but the name of the actual parameter that is passed. The technique to describe both parameter passing mechanisms is based on the lambda calculus.

In [7] another technique than that in [6] is presented where call-by-value and call-by
name semantics are expressed in the typed lambda calculus.

In our technique we actually use a call-by-value semantics, but the difference with the call-by-name semantics is hard to show in V-nus. This is due to the fact that only identifiers (variable names) can serve as actual parameters; other expressions are not allowed. Although we do not (yet) allow global variables in the procedure bodies, our technique is able to express dynamic scoping. As such, our semantics are 'stronger' than necessary for the current version of V-nus; the semantics are prepared to cope with global variables in some future version of V-nus.

In fact, program variables of every existing scope can be accessed in the V-nus program state. The location substate issues the current scope (see Section 3.3) which enables to address variables of the current scope (and in future versions of V-nus to address the 'previous scope', the 'global scope', etc. as well).

The index state gives rise to the construction of ordered index states and an order on index states, being two different things. An ordered index state has its elements ordered. It is needed to deal with the order of loop nests. An order on index states makes it possible to put index states 'in a row'. This is needed to cope with the order in which the index space of a loop is traced.

The semantics of language constructs other than parameter lists do not need much introduction. The definitions of the semantic functions are presented in a constructive style with hardly any forward references. Nevertheless, the functions are dependent on each other which causes the final definitions to have cross references. These final definitions can be looked up in Appendix B.

This chapter is organized as follows. In Section 4.2 we describe the construction of the initial states. Section 4.3 is a first step to give the semantics of Expression. Since there is some interaction between the definition of the semantics of expressions and definitions presented thereafter, the expression semantics are adjusted throughout this chapter. The View is described in Section 4.4. The semantics of functions and procedures and their parameter passing mechanism are described in Section 4.5. In Section 4.6 we revisit the semantics of Selection and simplify the corresponding semantic function. Finally, Section 4.7 is dedicated to the flow of control: the If, While, and Iteration.

4.1.1 Notational conventions

In presenting the denotational semantics the following notational conventions are used:

- The semantic functions that use V-nus syntax for input will have a name that starts with a calligraphical character. For instance, the function

  \[ \text{Set}((3, \ldots, n)) \]

  has the V-nus syntax \((3, \ldots, n)\) as input.

- The notation \(|X|\) is used to express the number of elements of the set \(X\). The notation is only used for finite sets.
• A short hand notation is used for the replacement function \( \approx \) with respect to a program state \( \sigma = (\sigma_V, \sigma_L) \). The expressions: \( \sigma \approx_X X \) and \( \sigma \approx_L Y \) are shortcuts for: \( (\sigma_V \times X, \sigma_L) \) and \( (\sigma_V, \sigma_L \times Y) \) respectively. For another arbitrary program state \( \sigma' = (\sigma'_V, \sigma'_L) \), the expression: \( \sigma \approx \sigma' \) is equal to \( \sigma_V \approx \sigma'_V \) and \( \sigma_L \approx \sigma'_L \).

Furthermore, we will use the term \textit{l-value} (of an assignment or view statement) to refer to the expression to which is assigned. The term \textit{r-value} refers to the expression that (is part of the expression that) is assigned to the l-value.

### 4.2 The initial states

Computing the meaning of a \textit{V-nus} program requires a declaration state, an index state, and a program state. In Chapter 3 the types of these states were introduced. Here, we will explain which states need to be used as the very first states for input to the meaning function.

The meaning function \( \mathcal{M} \) is of type: \( \text{Vnus} \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma \). So, given some \textit{V-nus} code, a declaration state, an index state, and a program state, the meaning function can compute the resulting program state after the \textit{V-nus} program has been executed. In case this \textit{V-nus} code is just a part of a \textit{V-nus} program, computing the meaning of this code starts with the states that are computed so far. In case the \textit{V-nus} code is the complete \textit{V-nus} program: program \textit{Initializations Declarations Statements SymbolTable} we need to agree on the initial declaration state, index state, and program state.

The initial states can be simple: \textit{V-nus} programs are self-contained such that these states can be empty. Since it is the beginning of a program, no index identifiers have been specified. This is why the initial index state is empty. The initial declaration state is also empty, because the declarations of a \textit{V-nus} program are all present in the \textit{Declarations} and in the \textit{SymbolTable}. Finally, the initial program state has no need for preset program variables, and thus is empty as well. These states are denoted as:

\[
\begin{align*}
\delta_{\text{Init}} &= \emptyset \\
\varphi_{\text{Init}} &= \emptyset \\
\sigma_{\text{Init}} &= (\emptyset, \emptyset)
\end{align*}
\]

In this thesis we focus on the meaning of executing \textit{V-nus} statements. That is, we will not go into computing the meaning of the initializations, the declarations, and the symbol table. We will simply assume that these are represented in the declaration state. Below, we will indicate how this declaration state is supposed to represent this kind of information. In order to do so, a notion of free memory is needed.

**Definition 4.2.1** The set of all locations that are garbage in a program state \( \sigma \) is denoted as:

\[
\text{FreeLoc}_\sigma
\]

such that \( \text{FreeLoc}_\sigma \subseteq \text{Loc} \) is valid for all program states \( \sigma \). \( \square \)
4.2. THE INITIAL STATES

We are interested in the following equation:

$$\mathcal{M}({\text{program Init Decs Stms ST}})(\delta_{Init})(\varphi_{Init})(\sigma_{Init}) = \mathcal{M}({\text{Stms}})(\delta_{Start})(\varphi_{Start})(\sigma_{Start})$$

where $\delta_{Start}$ and $\sigma_{Start}$ are still unknown. $\varphi_{Start}$ is the same as $\varphi_{Init}$. $\sigma_{Start}$ is the program state in which the Initializations are represented, and reflects the effect of some of the declarations. $\delta_{Start}$ is the declaration state in which the Declarations and the SymbolTable are represented. Computing these states is explained step-wise. Step 1 and 2 are used to get a proper representation of the symbol table. Step 3 explains the evaluation of the initializations, and step 4 processes the declarations.

**Step 1:** The SymbolTable needs to be represented in the declaration state. At this point we simply assume that this can be done. For an explanation of the symbol table entries we refer to Appendix A. So let’s say that representing the symbol table in the declaration state gives us the new declaration state $\delta'_{Init}$.

**Step 2:** Next, all identifiers that are declared as a data structure of type `shape` already need to occupy memory. That means, locations need to be reserved for these data structures. This is done by creating an $ip$ function in the view state for each of these data structures. An initial scope value needs to be introduced first. The special element $\ell$ is set to zero:

$$\sigma'_{Init}(\ell) = 0$$

Now, take into account all identifiers $Id$ for which $\delta'_{Init}(Id, s, \text{type}) = \text{shape}$, where $s$ can be any scope. Deduce the rank of $Id$; say, the rank of $Id$ is $r$; call $d$ the dimension of $Id$. Then let $\sigma'_{Init_v}$ be defined such that:

$$\sigma'_{Init_v}(Id, s) = \text{ip}$$

where $\{ s = \sigma'_{Init}(\ell), \text{ip} = \{(i, l_i) | i \in \mathbb{N}_d^{(0,r-1)}, l_i \in \text{FreeLoc}_{\sigma}\}$

**Step 3:** Compute the meaning of Initializations with the initial index state, the declaration state, and the program state constructed so far. The statements of Initializations are ordinary $V$-nus statements. Computing the meaning of these statements is explained in the remainder of this chapter. The meaning of having executed the Initializations is then reflected in the program state $\sigma''_{Init}$.

**Step 4:** Represent the Declarations in the declaration state, resulting in the new declaration state $\delta''_{Init}$. For an arbitrary function declaration:

```
function Id_F[p_0, \ldots, p_n] Id_R Statements
```
where the \( p_i \) are of type \( t_i \), and for every scope \( s \), we have:

\[
\begin{align*}
\delta''_{\text{Init}}(\text{Id}_F, s, \text{par}) &= [p_0, \ldots, p_n] \\
\delta''_{\text{Init}}(\text{Id}_F, s, \text{par}t) &= [t_0, \ldots, t_n] \\
\delta''_{\text{Init}}(\text{Id}_F, s, \text{result}) &= \text{Id}_R \\
\delta''_{\text{Init}}(\text{Id}_F, s, \text{stm}) &= \text{Statements}
\end{align*}
\]

For an arbitrary procedure declaration:

\[
\text{procedure } \text{Id}_P [p_0, \ldots, p_n] \text{ Statements}
\]

where the \( p_i \) are of type \( t_i \), and for every scope \( s \), we have:

\[
\begin{align*}
\delta''_{\text{Init}}(\text{Id}_P, s, \text{par}) &= [p_0, \ldots, p_n] \\
\delta''_{\text{Init}}(\text{Id}_P, s, \text{par}t) &= [t_0, \ldots, t_n] \\
\delta''_{\text{Init}}(\text{Id}_P, s, \text{stm}) &= \text{Statements}
\end{align*}
\]

In step 2, only data structures of type \text{shape} need to be represented by an \text{ip} function in the view state, since data structures of type \text{view} are dynamic. The latter do not occupy memory until the \text{ip} function is defined in a view statement. Step 3 is, in fact, recursively defined. The rest of this chapter needs to be known first, before this step can be taken. Step 4 puts the function and procedure declarations in the declaration state, such that this information can be used when the accompanying function call \text{eq} procedure call is encountered. Function and procedure declarations are not bound to the scope in which these are declared. That is why the scope \( s \) in step 4 can have any value.

The states \( \delta_{\text{Start}} \), \( \varphi_{\text{Start}} \), and \( \sigma_{\text{Start}} \) we are interested in are then defined as

\[
\delta_{\text{Start}} = \delta''_{\text{Init}} \\
\varphi_{\text{Start}} = \varphi_{\text{Init}} \\
\sigma_{\text{Start}} = (\sigma_v, \sigma_L)
\]

In Chapter 2 we mentioned that only ‘valid’ \( V\text{-nus} \) programs were to be considered. We extend this terminology to the states of our semantics. Consider a program fragment \( c \). The states \( \delta \), \( \varphi \), and \( \sigma \) are called ‘valid’ with respect to \( c \) if there exists a valid program \( V \in \text{Production} \), such that \( V = \text{program} \text{ Initializations} \text{ Declarations} \text{ statements} [c] \text{ SymbolTable} \) and \( M(V)(\delta_{\text{Init}})(\varphi_{\text{Init}})(\sigma_{\text{Init}}) = M(c)(\delta)(\varphi)(\sigma) \).

### 4.3 Expressions

Expressions are used in almost every part of the \( V\text{-nus} \) language. An expression is of type \text{Data} (as defined in Definition 3.2.6). Most of the operators that can be used to build an expression are well-known, and do not need additional explanation. However, expressions that represent an index of a data structure may use a special operator, called
the set selector. An index set is always constructed with the cardinality specification: \((i,n)\) such that the index identifier \(i\) represents the values \(0 \ldots n - 1\). Constructing references to elements of a data structure using expressions in \(i\) (without the set selector) restricts the referencing possibilities too much. Example 4.3.1 presents a situation that needs a set selector. The set selector allows the programmer to construct a temporary, non zero-based index set that may have 'holes'; i.e. it does not need to be contiguous. The generic form of an expression using the set selector \('!\) is: \(i ! \) SetExpression. The result of this expression is exactly one index, which is the \(i^{th}\) index of the index set SetExpression.

**Example 4.3.1** The need for a set selector.

Suppose a view \(V\) must be created on a data structure \(A\), such that all elements are viewed except the elements with index 4 to 6. Assume that \(A\) is one dimensional with rank \(n\) (\(n > 7\)). One will notice that it is impossible to create such a view without using the set selector.

The set selector solves this problem. The wanted view can be constructed with the \(V\)-nus term:

\[
\text{view } V \ (([(i, (n, -, 3))], (A, [(i, !, (\setminus, (4, .., 6))))]))
\]

The selector for the data structure \(A\) can be read as: take the \(i^{th}\) index of all indexes, but skip the index values 4, 5, and 6.

The meaning of an expression with the set selector is given in terms of the function \(\text{index}\). This function uses an index and a set of indexes. It puts the set of indexes in order and then counts 'from start' which element needs to be returned (see also Example 4.3.2).

**Definition 4.3.1** Let \(X \subseteq \mathbb{N}\). The function \(\text{index} : \mathbb{N} \times \mathcal{P}(\mathbb{N}) \rightarrow \mathbb{N}\) is defined in the
following way.

\[
\text{index}(i, X) = x_k \iff \begin{cases} \\
\exists f : \mathbb{N} \rightarrow \mathbb{N} \land f(i) = k \land \\
\forall j, x_{f(j)} \leq x_{f(j)} \land x_{f(j)} < x_{f(j+1)} \land \\
\overline{\exists x' \in X. x_{f(j)} < x' < x_{f(j+1)}} \end{cases}
\]

The set \(X\) in the definition of \(\text{index}\) represents a set of indexes. This set does not need to be a bounded set. It is just a collection of indexes. These kinds of sets can be constructed with the set operators of \(\text{V-nus}\). These consist of the unary operator \(\backslash\) which expresses difference, and the binary operators \(\cup\), \(\&\), and \(\ldots\) which express union, intersection, and succession respectively. The function \(f\) in the definition of \(\text{index}\) is needed to address the correct labels of the elements of \(X\). In Example 4.3.1 an instance of a set expression is given that excludes the set \{4, 5, 6\}. The construction of these index sets is formalized in the function \(\text{Set}\). This function returns an index set defined by the \(\text{V-nus}\) set expression. This index set may be infinite. The definition of \(\text{Set}\) uses a function \(E\), defined below, that returns a \(\text{Data}\) element for a given expression. An informal explanation of the meaning of \(E\) is given after the definition of \(\text{Set}\). Note: in the expression \((e, \ldots, e)\) below, the double dots are part of the \(\text{V-nus}\) language.

**Definition 4.3.2** Let \(se, se_1, se_2\) be arbitrary set expressions. Let \(e, e_1, e_2\) be arbitrary scalar expressions. The function \(\text{Set} : \mathcal{L}(-) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \mathcal{P}(-)\) is defined as

\[
\begin{align*}
\text{Set}((e_1, \ldots, e_2))(\delta)(\varphi)(\sigma) &= \{ \text{\(E(e_1)(\delta)(\varphi)(\sigma), \ldots, E(e_2)(\delta)(\varphi)(\sigma)\)} \} \\
\text{Set}(\backslash, se) &= \mathbb{N} \backslash \text{Set}(se) \\
\text{Set}(\backslash, e) &= \text{Set}(\backslash, (e, \ldots, e)) \\
\text{Set}((se_1, \backslash, se_2)) &= \text{Set}(se_1) \cup \text{Set}(se_2) \\
\text{Set}((se_1, \& , se_2)) &= \text{Set}(se_1) \cap \text{Set}(se_2)
\end{align*}
\]

The function call \(E(e)(\delta)(\varphi)(\sigma)\) computes the value of the expression \(e\) in the given declaration state \(\delta\), index state \(\varphi\), and program state \(\sigma\). This is formalized in the coming definition. For convenience, the function \(E\) is defined by using a function called \(B\), which is defined immediately thereafter in Definition 4.3.3. The function \(B\) gives a truth value to a boolean expression. As such, a call to \(B\) can be used as a conditional in function definitions.

The definition of \(E\) as it is given below is not complete. That is, in the rest of this chapter new kinds of expressions are added for which \(E\) will be defined. A complete definition of \(E\) can be found in Appendix B.
4.3. EXPRESSIONS

Preliminary definition Let $\otimes \in \{+, -, \ast, /, \text{DIV, MOD}\}$ represent a binary operator of the V-nus language, but not the operator $!$. Let $\in \{>, >=, <, =, <=, <, \text{AND, OR}\}$ represent a relational operator of the V-nus language. The function $\mathcal{E} : \mathcal{L}(Expression) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow Data$ is defined as follows.

\[
\begin{align*}
\mathcal{E}(d)(\delta)(\varphi)(\sigma) &= d 
\quad \text{if } d \in Data_0 \\
\mathcal{E}(i)(\delta)(\varphi)(\sigma) &= \varphi(i) 
\quad \text{if } i \in I
\end{align*}
\]

\[
\mathcal{E}(\text{shape} [e_1, \ldots, e_n])(\delta)(\varphi)(\sigma) = (\mathcal{E}(e_1)(\delta)(\varphi)(\sigma), \ldots, \mathcal{E}(e_n)(\delta)(\varphi)(\sigma))
\]

\[
\mathcal{E}((i, !, e))(\delta)(\varphi)(\sigma) = \text{index}(\mathcal{E}(i)(\delta)(\varphi)(\sigma), \text{Set}(e)(\delta)(\varphi)(\sigma))
\]

\[
\begin{align*}
\mathcal{E}((e_1, \in, e_2)) &= \mathcal{B}((e_1, \in, e_2)) \\
\mathcal{E}(\text{true}) &= \mathcal{B}(\text{true}) \\
\mathcal{E}(\text{false}) &= \mathcal{B}(\text{false})
\end{align*}
\]

The expression $\text{shape} [\ldots]$ is a way to denote a data structure in a single step. Note that a data structure $\text{shape} [e]$ with only one element is not distinguished from the bare element $e$.

**Definition 4.3.3** The function $\mathcal{B} : \mathcal{L}(Expression) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \mathcal{B}$ is defined as

\[
\mathcal{B}(\text{true})(\delta)(\varphi)(\sigma) = \text{true}
\]

\[
\mathcal{B}(\text{false})(\delta)(\varphi)(\sigma) = \text{false}
\]

\[
\mathcal{B}((e_1, \in, e_2)) = \mathcal{E}(e_1)\overset{\in}{\mathcal{E}}(e_2)
\]

It is easy to see that the functions $\mathcal{E}$ and $\mathcal{B}$ do not induce an infinite recursion; in other words, $\mathcal{E}$ and $\mathcal{B}$ are well-defined. Assume that $e$ is an arbitrary expression. Then we distinguish three kinds of function applications that are used in the definition for $\mathcal{E}(e)$.

1. $\mathcal{E}(e)$ is neither defined in terms of $\mathcal{E}$ nor in terms of $\mathcal{B}$. Trivial.

2. $\mathcal{E}(e)$ is defined in terms of $\mathcal{E}$. Then the recursive call to $\mathcal{E}$ is given only a part of $e$ (not $e$ itself). It is clear that no infinite recursion occurs in this way.

3. $\mathcal{E}(e)$ is defined in terms of $\mathcal{B}$. Then the given argument to $\mathcal{B}$ is equal to $e$. The computation of $\mathcal{B}(\text{true})$ and $\mathcal{B}(\text{false})$ will stop by definition. For all other calls, $\mathcal{B}(e)$ is defined in terms of $\mathcal{E}$, where $\mathcal{E}$ is given an argument that is only a part of $e$ (and not $e$ itself).

So, $\mathcal{E}$ and $\mathcal{B}$ are well-defined. This reasoning will stay valid when the definition of $\mathcal{E}$ is extended in the rest of this chapter.
Example 4.3.2 The set selector revisited.
Consider again the view statement of Example 4.3.1. The value of the selector for A in case i represents 2 or 5 is computed as follows.

\[
\text{Set}((\emptyset, (4, \ldots, 6)))(\delta)(\varphi)(\sigma) = \\
\mathbb{N}\setminus\text{Set}((4, \ldots, 6)) = \\
\mathbb{N}\setminus\{(4,5,6)\}
\]

For i representing 2 we have: \(\mathcal{E}(i)(\delta)(\varphi)(\sigma) = \varphi(i) = 2\). So,

\[
\mathcal{E}((i, \emptyset, (4, \ldots, 6)))(\delta)(\varphi)(\sigma) = \\
\text{index}(2, \mathbb{N}\setminus\{4,5,6\}) = \\
2
\]

In words: the second index in \(\mathbb{N}\setminus\{4,5,6\}\) is 2 (where 0 is the zero-th index). For i representing 5 we have \(\text{index}(5, \mathbb{N}\setminus\{4,5,6\}) = 8\). The fifth index is 8 because indexes 4, 5, and 6 must be skipped.

4.3.1 Selections

The syntax of a Selection is \(\text{Expression}, \text{Selectors}\) (see also Section 2.3.6. It means that Selectors represent an index for the data structure Expression, which refers to a scalar (directly, or via a location). To formalize this, we introduce a function \(I\) that computes the index that is described by the Selectors.

Definition 4.3.4 Let \(s_0, \ldots, s_n \in \mathcal{L}(\text{Selector})\) each represent a selector. The function \(I : \mathcal{L}(\text{Selectors}) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow bs\), for a certain bounded set \(bs \in BSS\), is defined as

\[
I([s_0, \ldots, s_n])(\delta)(\varphi)(\sigma) = (\mathcal{E}(s_0)(\delta)(\varphi)(\sigma), \ldots, \mathcal{E}(s_n)(\delta)(\varphi)(\sigma))
\]

Note that \(I([\ ])(\delta)(\varphi)(\sigma) = ()\) for all states.

The selection as l-value

When the Selection is used as an l-value in the assignment, we are interested in the location that is referenced by a Selection, such that new data can be put in this location. For this purpose we define a function \(\text{Loc}\) that computes the location in which the value of Selection is recorded. At this moment only the syntax: \((Id, Selectors)\) is taken into account. Another form of Selection uses a function call. Using such a kind of Selection is described in Section 4.5.2. A complete definition of \(\text{Loc}\) can be found in Appendix B.
4.3. EXPRESSIONS

Preliminary definition  The function $\text{Loc} : \mathcal{L}(\text{Selection}) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \text{Loc}$ is defined as

$$\text{Loc}((\text{Id}, \text{Selectors}))(\delta)(\varphi)(\sigma) = \sigma_{\text{V}}(\text{Id}, s)(i)$$

where

$$\begin{cases} s &= \sigma_{\text{L}}(\ell) \\ i &= I(\text{Selectors})(\delta)(\varphi)(\sigma) \end{cases}$$

Remember that $s$ represents the current scope. The function call $\sigma_{\text{V}}(\text{Id}, s)$ returns the 
i$p$ function that represents the data structure $\text{Id}$. The call to this 
i$p$ function with the index $i$ determines the corresponding location.

The way locations are used to assign to is postponed until Section 4.4.3, where the assignment is formalized. A trivial example using $\text{Loc}$ is given in Example 4.3.3.

---

Example 4.3.3  Computing the locations of a view.
Consider a $2 \times 2$ matrix $A$. Say, the element $A_{i,j}$ is stored in location $l_{i,j}$, for $i, j \in \{0, 1\}$. With $\text{Loc}$ we can use this relation between data and their locations. The location of element $A_{0,1}$ is computed in valid states $\delta$, $\varphi$, and $\sigma$ as follows.

$$\text{Loc}((A, [0,1]))(\delta)(\varphi)(\sigma) = \sigma_{\text{V}}(A, \sigma_{\text{L}}(\ell))(0,1) = l_{0,1}$$

---

The selection as r-value

The meaning of a $\text{Selection}$ as an r-value is described by the function $\mathcal{E}$. In this case we are interested in the data that is represented by the $\text{Selection}$. Again, the semantics of a $\text{Selection}$ as r-value is not described completely. In the first place, we postpone the use of function calls until Section 4.5.2. Secondly, we cannot yet present the generic semantics for a $\text{Selection}$; other extensions to $\mathcal{E}$ are required first. Below, two preliminary extensions are presented in order to cope with the forms: $(\text{Id}, \text{Selectors})$ and $(\text{Shape}, \text{Selectors})$; e.g. $\text{Shape}$ represents shape $[e_0, \ldots, e_n]$. In Section 4.6 we revisit the semantics of a $\text{Selection}$ as an r-value, and present one definition for the complete class of $\text{Selection}$.

For the construct $(\text{Id}, \text{Selectors})$ it is assumed that $\sigma_{\text{V}}(\text{Id}, s)$ is defined for the current scope $s$, and is thus represented by an 
i$p$ function. This 
i$p$ function returns a location for a specific index. So, we also assume that the locations that are referenced by this 
i$p$ function contain data. Recording such an 
i$p$ function in the program state is formalized in Section 4.4.

The meaning of $(\text{Id}, \text{Selectors})$ is formalized with the following preliminary extension to the function $\mathcal{E}$:

$$\mathcal{E}((\text{Id}, \text{Selectors}))(\delta)(\varphi)(\sigma) = \sigma_{\text{L}}(\text{Loc}((\text{Id}, \text{Selectors}))(\delta)(\varphi)(\sigma))$$
The expression *Shape* can also be used as a data source to select from. The function $E$ is preliminarily defined for the form $(Shape, Selectors)$ as follows.

$$E((Shape, Selectors))(\delta)(\varphi)(\sigma) = d_i$$

where

$$\begin{align*}
\{ d &= E(Shape)(\delta)(\varphi)(\sigma) \\
i &= I(Selectors)(\delta)(\varphi)(\sigma)
\end{align*}$$

### 4.4 Views

The view statement is the mechanism in *V-nus* to assign an *ip* function to a data structure of type *view*. View statements have a 'declarative character'; i.e. once a data structure of type *view* is represented by an *ip* function due to a view statement, this *ip* function cannot be changed. It is only possible to assign a new *ip* function to the view identifier with another view statement. Of course, the data of a view can be changed with an assignment.

The assignment statement assigns data to locations. Using an assignment does not change the *ip* function that is used to direct to the appropriate locations.

The term 'view' is, in fact, overloaded in this thesis. In the language *V-nus* a type *view* exists. Declaring an identifier of type *view* means that it can be used as an l-value in a view statement. An identifier of any other type is not allowed to be used as an l-value in a view statement.

A different meaning of the term 'view' is used in a semantical context. In our semantic model we consider each data structure as a 'view' on the set of locations. This is a wider group of data structures than those that are typed as *view* in the *V-nus* language, since data structures of type *shape* are incorporated too. From now on, we will use the term 'view' as it is used in the semantical context. That implies: the term 'view' and the term 'data structure' mean one and the same. In case the other meaning is meant, we will explicitly call it a 'data structure of type *view*'.

Before introducing the denotational semantics of views, view statements, and assignments, we need another representation of index states. In Example 3.6.2 a loop nest has been presented such that the index variables $i_1$ and $i_2$ are in the outermost loop, and $j$ is in the innermost loop. In that case we had to deal with vectors of the form: $(i_1, i_2, j)$. An index state does not keep the ordering between the index variables, which is why another representation is needed where the ordering is preserved.

### 4.4.1 The view expression

We need a notion of an *ordered index state*. An ordered index state is a representation of an index state in the form of a tuple. A generic index state $\varphi$ can be represented by $
\{(x_0, y_0), \ldots, (x_n, y_n)\}$. An *ordered representation* of the index state $\varphi$ will be denoted as a tuple, for instance $(\{(x_0, y_0), \ldots, (x_n, y_n)\})$ is an ordered representation of $\varphi$ (called an
ordered index state). Of course, an index state allows more than one ordered representation in general.

First, we define how to denote a set of tuples with elements of a certain given set.

**Definition 4.4.1** For an arbitrary set $X$ the set $X|_n$, for $n \in \mathbb{N}$, is defined as follows.

$$X|_n = \{(x_0, \ldots, x_{n-1}) | x_i \in X, \ 0 \leq i \leq n-1\}$$

$X|_n$ is the set of $n$-tuples over $X$. □

An ordered representation of a set $X$ is a tuple that has as many elements as $X$, such that each element of $X$ is also present in the ordered representation.

**Definition 4.4.2** Let $X$ be an arbitrary finite set, say $|X| = n$. Let $t = (t_0, \ldots, t_{n-1}) \in X|_n$.

$t$ is an **ordered representation** of $X$

$$\iff \forall x \in X \ \exists i \in \{0, \ldots, n-1\} : x = t_i$$

□

With these definitions we can construct an ordered index state as an ordered representation of an index state. Example 4.4.1 may make this more clear.

---

**Example 4.4.1** An index state and some ordered representations.

Consider the index state $\varphi \in \Phi$ defined by $\varphi = \{(i, 2), (j, 3), (k, 7)\}$. The tuples $((i, 2), (j, 3), (k, 7))$ and $((j, 3), (i, 2), (k, 7))$ are examples of ordered representations of $\varphi$. The tuple $((i, 2), (i, 2), (j, 3))$ is an element of $\varphi|_3$, but is not an ordered representation, since $(k, 7)$ is missing.

---

So, what use is this for the semantics of a view? Ordered index states are necessary to define the semantics of Cardinalities, and are required when we have to deal with loop constructs, where it is possible to go through a (nested) index space in a predefined order (such as the loop nest of Example 3.6.2). This is clarified in Section 4.7.

The expression View is of the form (Cardinalities, Selection) and is represented by an ip function. The Cardinalities specify the domain of the ip function, which is defined by the function $\mathcal{D}P$ (domain propagation).

**Definition 4.4.3** Let $T_k = \{t | t$ is an ordered representation of an arbitrary $\varphi$, $|\varphi| = k\}$ be the set of all ordered index states of length $k$. Let $i_0, \ldots, i_{k-1} \in \Pi$ be index identifiers. Let $e_0, \ldots, e_{k-1} \in \mathcal{L}(Expression)$ be scalar expressions. The function $\mathcal{D}P$:
\[ \mathcal{L}(\text{Cardinalities}) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow T_k \text{ is defined as} \]

\[
\mathcal{D}P([[(i_0, e_0), \ldots, (i_{k-1}, e_{k-1})]])(\delta)(\varphi)(\sigma) =
\begin{cases}
\{(i_0, n_0), \ldots, (i_{k-1}, n_{k-1})\} & \text{if } k \geq 1 \\
\{()\} & \text{otherwise}
\end{cases}
\]

where
\[
\begin{align*}
e_0' &= \mathcal{E}(e_0)(\delta)(\varphi)(\sigma) \\
&\vdots \\
e_{k-1}' &= \mathcal{E}(e_{k-1})(\delta)(\varphi)(\sigma)
\end{align*}
\]

and where \( 0 \) represents the zero vector of length \( k \).

So, for an empty cardinality list we have \( \mathcal{D}P([()](\delta)(\varphi)(\sigma) = \{()\} \).

The \( \mathcal{D}P \) function creates a set of ordered index states. It is the intention to pick an ordered index state out of this set, and transform this ordered index state into the original index state. This transformation is a simple computation for which we introduce a special notation.

**Definition 4.4.4** Consider an arbitrary tuple \( t = (t_0, \ldots, t_n) \). The minimal set \( \langle t \rangle \) to construct \( t \) is defined as

\[ \langle t \rangle = \{t_0, \ldots, t_n\} \]

and for a set \( T = \{x_0, \ldots, x_n\} \) of tuples, \( \langle T \rangle \) is defined as

\[ \langle T \rangle = \{(x_0), \ldots, (x_n)\} \]

Remark: \( \langle () \rangle = \emptyset \). Note that for an arbitrary tuple \( t = (t_0, \ldots, t_n) \) it is possible to have: \( |\langle t \rangle| < n + 1 \).

The relationship between ordered representations and minimal sets is presented in a theorem.

**Theorem 4.1** Say \( \text{dim}(t) \) is the dimension of \( t \). Then

\[
t \text{ is an ordered representation of } \varphi \\
\iff \\
\varphi \text{ is the minimal set to construct } t \text{ (} \varphi = \langle t \rangle \text{), and } |\varphi| = \text{dim}(t)\]

\[]
4.4. VIEWS

Proof. We have \( i \neq j \Rightarrow \text{proj}_i(t) \neq \text{proj}_j(t) \).

1. Assume \( t \) is an ordered representation of \( \varphi \). By Definition 4.4.2 we have: \( \forall x \in \varphi \exists i. x = t_i \). Thus \( \varphi \) is the minimal set to construct \( t \).

2. Assume \( \varphi \) is the minimal set to construct \( t \) and \( |\varphi| = \text{dim}(t) \). By Definition 4.4.4 we have: \( \langle t \rangle = \varphi \); say \( \langle\{t_0, \ldots, t_n\}\rangle = \varphi = \{t_0, \ldots, t_n\} \). Then \( \forall x \in \varphi \exists i. x = t_i \). Thus \( t \) is an ordered representation of \( \varphi \).

We can now define the meaning of a View. This meaning is given in the form of an ip function. A function \( IP \) is defined that returns an ip function that corresponds with the given View. The function \( IP \) is extended for function calls (representing a view) in Section 4.5. A complete definition of \( IP \) can be found in Appendix B.

Preliminary definition. Let \( \text{Cardinalities} \) be a cardinality list for the index identifiers \( i_0, \ldots, i_n \). The function \( IP : \mathcal{L}(\text{Expression}) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \text{IPS} \) is defined as

\[
IP((\text{Cardinalities}, \text{Selection}))(\delta)(\varphi)(\sigma) = \\
\{(\varphi'(i_0), \ldots, \varphi'(i_n)), \text{Loc}(\text{Selection})(\delta)(\varphi \triangleleft \varphi')(\sigma)\} \\
\varphi' \in \{\text{DP}(\text{Cardinalities})(\delta)(\varphi)(\sigma)\}
\]

The definition of \( IP \) is clarified with Example 4.4.2.

Example 4.4.2 Deriving an ip function from a View.
Assume the existence of a data structure \( A \) where \( A_1 \) and \( A_2 \) are recorded in the locations \( l_1 \) and \( l_2 \) respectively. Now, consider the following View expression:

\( \langle\{i, 2\} \rangle, (A, \langle i, +, 1 \rangle) \rangle \)

It can easily be seen that the corresponding ip function should map the index values 0 and 1 to the locations of \( A_1 \) and \( A_2 \). Assume that \( \sigma \) is a valid program state for the mentioned context, and for simplicity, assume that the index state is empty (i.e. the View does not occur in a loop nest). The declaration state \( \delta \) is arbitrary since it is not used. Well, this ip function can be derived as follows:

\[
IP(\langle\{i, 2\} \rangle, (A, \langle i, +, 1 \rangle) \rangle)(\delta)(\varphi)(\sigma) = \\
\{(\varphi'(i), \text{Loc}(A, \langle i, +, 1 \rangle))(\delta)(\varphi \triangleleft \varphi')(\sigma)\} \quad \varphi' \in \{\{(i, 0), \{(i, 1)\}\}\} \}
\]

\[
\{(\varphi'(i), \text{Loc}(A, \langle i, +, 1 \rangle))(\delta)(\varphi'(\sigma))\} \quad \varphi' \in \{\{(i, 0), \{(i, 1)\}\}\} = \\
\{(0, l_1), (1, l_2)\}
\]

which is the wanted ip function.
When a View is used in an Expression the function \( \mathcal{E} \) needs to be defined for a View. Therefore, another extension to this function \( \mathcal{E} \) is the following.

\[
\mathcal{E}(\text{View})(\delta)(\varphi)(\sigma) = \text{ip2data}(ip)(\sigma)
\]

where \( ip = \mathcal{I}(\text{View})(\delta)(\varphi)(\sigma) \)

### 4.4.2 The view statement

Finally, the meaning of a ViewStatement can be expressed in a rather straightforward way. As mentioned at the start of this chapter, we use the meaning function \( \mathcal{M} \) to define the semantics of V-nus statements. The complete definition of this meaning function will be given in Appendix B. The denotational semantics of a ViewStatement can be formalized in the following way.

\[
\mathcal{M}(\text{ViewStatement})(\delta)(\varphi)(\sigma) = \sigma \bowtie \sigma'_{\text{V}}.
\]

\[
\begin{align*}
\text{ViewStatement} & = \text{view Id View} \\
\sigma'_{\text{V}} & = \{((\text{Id}, s), ip)\} \\
ip & = \mathcal{I}(\text{View})(\delta)(\varphi)(\sigma) \\
s & = \sigma_{L}(\ell)
\end{align*}
\]

The function call \( \mathcal{M}(\text{ViewStatement})(\delta)(\varphi)(\sigma) \) returns a new program state in which the \( ip \) function for the View is recorded. In case a view \( Id \) in scope \( s \) was already declared by some previous view statement, it is completely redefined when another view statement is encountered for the same view \( Id \) in scope \( s \).

### 4.4.3 The assignment

Only one kind of assignment is possible in V-nus, which is the scalar assignment. That means that only a scalar can be assigned to a variable. Strictly seen, the view statement is an assignment too, since an \( ip \) function is assigned to a variable of type \textit{view}. However, a view statement will not be referred to as an assignment, and with 'assignment' we only mean those statements that are defined by the non-terminal Assignment.

The meaning of assignment Selection Expression is straightforward. The value of Expression needs to be stored in the location of Selection. The value of an expression is determined with the function \( \mathcal{E} \); the location of a data element is determined with the function Loc. Therefore, the meaning of an assignment is as follows.

\[
\mathcal{M}(\text{Assignment})(\delta)(\varphi)(\sigma) = \sigma \bowtie \sigma'_{L}
\]

where \( \begin{align*}
\text{Assignment} & = \text{assignment Selection Expression} \\
\sigma'_{L} & = \{((\text{Loc}(\text{Selection})(\delta)(\varphi)(\sigma), \mathcal{E}(\text{Expression})(\delta)(\varphi)(\sigma))\}
\end{align*} \)

It is also possible to assign to the return value of a function call. This only makes sense when the return value is of type \textit{view}, since this return value becomes garbage immediately.
after the function call has finished. Assigning to a return value of type view means that an
assignment is performed to the data element that is viewed. This data element still exists
after the function call has finished. The semantics of using function calls is described in
the next section.

4.5 Functions and procedures

4.5.1 Parameter passing

Parameter passing is an important mechanism to exchange data between different scopes.
In V-nus all parameters are passed ‘by value’. The type of this parameter determines
what ‘the value’ of the parameter is. The type of a parameter is specified in the formal
parameter list, which requires the corresponding actual parameters to be of the same type.

A formal parameter is (the name of) the variable that is specified in the parameter
list FormalParameters of the procedure or function declaration. A formal parameter has
a type, but does not represent an existing data structure. An actual parameter is (the
name of) the variable that becomes local to the procedure or function, and represents an
actual data structure. Which data structure it represents is specified by the accompanying
procedure or function call.

When a data structure of type view is passed to a procedure, its ip function is taken
as the value that can be used in the body. At the time this thesis is written, it is not
allowed by the V-nus compiler to pass view-typed parameters to functions. An assignment
to the passed ip function would have its effect in the referenced locations. That causes
the effect of the assignment to become visible in the outer scope, which leaves us with
a non-referentially transparent function. Due to considerable time and space costs in the
compiled program this feature has been disabled in the language. The semantic description
has no need for this restriction. That means that it is possible to pass a view typed data
structure to a function, assign to this data structure in the function body, and still preserve
referential transparency. This is made concrete in Section 4.5.2.

Using view-typed parameters in procedures, unlike functions, gives the programmer the
possibility to simulate parameter passing ‘by reference’. However, modifications to the ip
function itself are not visible in the outer scope. In Example 4.5.1 it is shown how to pass
a data reference to a procedure.

The value of data structures of any other type (when passed to a function or procedure)
is the data element (of type Data) it represents. For instance, if E is of type shape, the
data structure d ∈ Data that is represented by E is seen as ‘the value of E’. So, d is copied
to new locations and a new ip function is constructed that refers to the copy of d.
Example 4.5.1 Parameter passing 'by value' and 'by reference'.
Passing a data structure \( A \) by reference to a procedure \( P_2 \) can be simulated by creating a view \( V \) on \( A \) first, and then pass \( V \) to \( P_2 \). So, assume \( A \) is a data structure (of type \textit{shape}) with rank \( n \), and \( V \) is a data structure of type \textit{view}. Let \( P_2 \) be a procedure that requires a \textit{view} typed parameter \( X \) and assigns the value 13 to \( X \); let \( P_1 \) be the same as \( P_2 \) except that it requires a \textit{shape} typed parameter. Now consider the following \textit{V-nus} code:

\[
\begin{align*}
\text{assignment} & \; (A, [0]) \; 7, \\
\text{view} & \; V \; ([i], \; ([i, n]), \; (A, \; [i])), \\
\text{procedurecall} & \; P_1 \; [A], \\
\text{procedurecall} & \; P_2 \; [V]
\end{align*}
\]

After \( P_1 \) has been called (and executed) \( A_0 \) is still 7. After \( P_2 \) has been called (and executed) \( A_0 \) has been changed to 13.

When executing the body of a procedure or a function, a new scope must be created in which the execution takes place. Afterwards, the original scope must become active again. Creating a new scope means incrementing the scope number; leaving a scope means decrementing the scope number. Since this scope number is always needed when accessing a program variable (of type \( VI \)) there is no name conflict between equally named variables in different scopes.

For creating and leaving a scope, the functions \textit{NewScope} and \textit{LeaveScope} are defined.

Definition 4.5.1 The function \textit{NewScope} : \( \Sigma \rightarrow \Sigma \) is defined as

\[
\text{NewScope}(\sigma) = \sigma \triangleleft L (\ell, \sigma_L(\ell) + 1)
\]

The function \textit{LeaveScope} : \( \Sigma \rightarrow \Sigma \) is defined as

\[
\text{LeaveScope}(\sigma) = \begin{cases} 
\sigma \triangleleft L (\ell, \sigma_L(\ell) - 1) & \text{if } \sigma_L(\ell) > 0 \\
\bot & \text{otherwise}
\end{cases}
\]

We will now formalize this parameter passing in the denotational semantics. We introduce two functions: \textit{PassIp} and \textit{PassData} which modify the program state such that the \textit{ip} function and the data element is passed to the new scope, respectively. The function \textit{PassIp} makes the given \textit{ip} function accessible through the given view identifier.

Definition 4.5.2 The function \textit{PassIp} : \((VI \times IPS) \rightarrow \Sigma \rightarrow \Sigma \) is defined as follows.

\[
\text{PassIp}(Id, ip)(\sigma) = \sigma \triangleleft V \sigma_V'
\]

where \[
\begin{align*}
\sigma_V' &= \{((Id, s), ip)\} \\
s &= \sigma_L(\ell)
\end{align*}
\]
4.5. FUNCTIONS AND PROCEDURES

The function PassData saves the given data element in free locations, creates a new ip function that refers to the data element, and makes this ip function accessible through the given view identifier.

**Definition 4.5.3** The function \( \text{PassData} : (VI \times Data) \rightarrow \Sigma \rightarrow \Sigma \) is defined as follows.

\[
\text{PassData}(Id, d)(\sigma) = \sigma' \triangleleft_L \sigma''_L
\]

\[
\begin{align*}
bs_1 &= \{ i \mid d_i \in \text{Data}_0 \} \\
|bs_2| &= |bs_1| \quad \exists bs_2 \subseteq \text{FreeLoc}_\sigma, \quad |bs_2| \quad \sigma''_L = \{ (l_i, d_i) \mid i \in bs_1, l_i \in bs_2, \forall i, j \in bs_1, i \neq j \Rightarrow l_i \neq l_j \} \\
\text{ip} &= \text{data2ip}(d, bs_2)(\sigma \triangleleft_L \sigma''_L) \\
\sigma' &= \text{PassIp}(Id, ip)(\sigma)
\end{align*}
\]

The bounded set \( bs_1 \) is an index set for the data element \( d \). The bounded set \( bs_2 \) is an equally sized set of free locations. In practice, \( bs_2 \) need not necessarily be a set with 'no holes'; it can be any set of free locations as long as it has the same number of elements as \( bs_1 \). Then, \( \sigma''_L \) is a location substate in which the data element \( d \) is stored in the locations of \( bs_2 \). Next, \( \text{ip} \) becomes an ip function for this data element. And finally, this ip function can be accessed by the variable name \( Id \) in the current scope.

Passing data via parameters means that the 'complete' referenced data structure is passed. As can be seen in the V-nus grammar only a single (view) identifier can be used to specify a data structure in a parameter list. As such, the complete corresponding ip function is referenced by a view identifier in an actual parameter list. This choice in constructing the grammar is based on the following considerations. Suppose a Selection is passed. In the procedure body this view-part could be redirected, such that it views a data structure different from the one it originally viewed. Exiting this procedure body would cause a conflict, since it is not allowed to have the same view on two different data structures. For a data structure typed as shape a different argument can be given. In future versions of V-nus it will be possible to 'resize' a data structure; that is, one can change the cardinality of a data structure. In that case a conflict occurs when only a part of a data structure is 'resized'.

Normally, Selectors are needed in combination with an identifier to indicate the elements of the ip function domain. In order to cope with single identifiers in an actual parameter list we extend the function \( \mathcal{E} \) in the following way.

\[
\mathcal{E}(Id)(\varphi)(\sigma) = \text{ip2data}(ip)(\sigma) \quad \text{if } Id \in VI
\]

\[
\text{where } \left \{ \begin{array}{l}
\text{ip} = \sigma_V(Id, s) \\
s = \sigma_L(\ell)
\end{array} \right.
\]
One aspect of passing data to procedures and functions has not been described yet: detecting whether an \textit{ip} function has to be passed, or a data element. The function $Pass$ uses three \textit{V-nus} terms (a formal parameter name, an actual parameter name, and a type name) and calls the appropriate function $PassIp$ or $PassData$.

**Definition 4.5.4** The function $Pass : (VI \times VI \times \mathcal{L}(Type)) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma$ is defined as

\[
Pass(Id_f,Id_a,t)(\delta)(\varphi)(\sigma) = \sigma' \\
\begin{cases} 
PassIp(Id_f,ip)(\sigma) & \text{if } t = \text{view} \\
PassData(Id_f,d)(\sigma) & \text{otherwise}
\end{cases}
\]

where

\[
\begin{align*}
ip &= \sigma_V(Id_a,s) \\
d &= \mathcal{E}(Id_a)(\delta)(\varphi)(\sigma) \\
s &= \sigma_L(\ell)
\end{align*}
\]

\[
\square
\]

### 4.5.2 The function call

A function call is of type \textit{Expression}. but can be used both as r-value and l-value. In case it serves as an r-value the meaning of a function call is the data element it returns. When it is used as an l-value the meaning of this function call is the set of locations in which the returned data element is stored. As such it is possible to assign to the return value of a function call. However, if the returned data structure is not of type \textit{view}, the locations of this data structure become garbage when the encompassing statement ends in which the function was called. Because, at the time this statement has ended, there is no reference anymore to the locations of the return value. When a view is returned that refers to a data structure that is ‘known’ outside the function call, assignments to the return value remain accessible after the encompassing statement has ended.

Note that there may be parameters that refer to data that exist in the outer scope of a function. Assignments within the function body to such a parameter have no effect on the data in the outer scope. The assignments have a temporary (local) character.

The type of the actual parameters of a function call need to have the same type as given to the formal parameters. So, type casting is not possible in \textit{V-nus} when matching actual parameters with the formal parameters.

Like views (meaning an arbitrary data structure), a function call is represented with an \textit{ip} function. Therefore, we extend the function $\mathcal{I}P$ for function calls, which was already defined for a \textit{View}. The complete definition of this function can be found in Appendix B.
\[\text{IP(functioncall } \text{Id}_F [e_0, \ldots, e_n])((\delta)(\phi))(\sigma) = ip\]

where \[
\begin{align*}
\sigma_0 &= \text{NewScope}(\sigma) \\
\sigma_1 &= \sigma_0 \downarrow_{\text{Pass}(p_0, e_0, t_0)(\delta)(\phi)(\sigma_0)} \downarrow_{\ldots} \\
&\quad \downarrow_{\text{Pass}(p_n, e_n, t_n)(\delta)(\phi)(\sigma_0)} \\
\sigma' &= \mathcal{M}(\text{Statements})(\delta)(\emptyset)(\sigma_1) \\
ip &= \sigma'_v(\text{Id}_R, s) \\
s &= \sigma_{0_L}(\ell) \\
[p_0, \ldots, p_n] &= \delta(\text{Id}_F, s, \text{par}) \\
[t_0, \ldots, t_n] &= \delta(\text{Id}_F, s, \text{partype}) \\
\text{Statements} &= \delta(\text{Id}_F, s, \text{stm}) \\
\text{Id}_R &= \delta(\text{Id}_F, s, \text{result})
\end{align*}\]

Executing the body of a function requires a new scope first, returning the program state \(\sigma_0\). Then all given parameters are passed, returning the state \(\sigma_1\). The \(p_i\) are the names of the formal parameters of the function. The \(t_i\) are the corresponding types of the formal parameters. Then the function body \text{Statements} can be executed which results in the program state \(\sigma'\). The index variables of the outer scope are not visible in the body, which is why the meaning is computed with an empty index state. The return variable \(\text{Id}_R\) got its value somewhere during the execution of \text{Statements}. So finally, the result of the function \text{IP} is the \(ip\) function of the return variable, which is recorded in the substate \(\sigma'_v\).

Note that the program state \(\sigma'\) is lost when the function call returns its value. That means that assignments in the function body have no effect in the outer scope. So, \text{view} typed parameters could be used and assigned to within the function body, and still let the function be referentially transparent.

The function call as l-value

For using the result of a function call as a data structure to assign to, we recall the function \(\text{Loc}\). This function determines the locations of a given \text{Selection}. Until now, only selections of the form: \((\text{Id}, \text{Selectors})\) can be input to this function. Instead of the view identifier \(\text{Id}\) it is also possible to denote a function call. The function \(\text{Loc}\) is therefore extended with the following definition.

\[\text{Loc}((\text{FunctionCall}, \text{Selectors}))((\delta)(\phi))(\sigma) = ip(i)\]

where \[
\begin{align*}
ip &= \text{IP}(\text{FunctionCall})(\delta)(\phi)(\sigma) \\
i &= \text{I}(\text{Selectors})(\delta)(\phi)(\sigma)
\end{align*}\]

With this definition it is possible to compute the locations of the return value of a function call. As a result, the semantics for an assignment (see Section 4.4.3) are now defined for function calls as l-value too.
The function call as r-value

When a function call is used as an r-value it denotes a data element. This can be simply retrieved by using the function \( \text{ip2data} \) of Definition 3.6.4. So, it is rather straightforward to extend \( \mathcal{E} \) with

\[
\mathcal{E}(\text{FunctionCall})(\delta)(\varphi)(\sigma) = \text{ip2data}(\text{ip})(\sigma) \\
\text{where } \text{ip} = \text{IP}(\text{FunctionCall})(\delta)(\varphi)(\sigma)
\]

4.5.3 The procedure call

A procedure call belongs to the class of Statements. Therefore, the meaning of a procedure call is represented by a change in the program state. The ‘mechanism’ to compute the meaning resembles that of the function call. The main difference is the result of the semantic function that computes this meaning; it returns a program state instead of an \( \text{ip} \) function. The returned program state is the (possibly changed) state after the procedure body instance has been executed.

For convenience we require that no global variables can be used within a procedure. This will not decrease the expressibility of procedures with respect to procedures in which one can use global variables. Global variables can be passed ‘by reference’ in the \( V-nus \) procedures (see Section 4.5.1).

We formalize another definition for the meaning function \( \mathcal{M} \) that describes the meaning of a procedure call as follows.

\[
\mathcal{M}(\text{procedurecall } \text{Id}_p \, [e_0, \ldots, e_n])(\delta)(\varphi)(\sigma) = \sigma' \\
\sigma_0 = \text{NewScope}(\sigma) \\
\sigma_1 = \sigma_0 \triangleleft \text{Pass}(p_0, e_0, t_0)(\delta)(\varphi)(\sigma_0) \triangleleft \ldots \triangleleft \text{Pass}(p_n, e_n, t_n)(\delta)(\varphi)(\sigma_0) \\
\sigma_2 = \mathcal{M}(\text{Statements})(\delta)(\varphi)(\sigma_1) \\
\sigma' = \text{LeaveScope}(\sigma_2) \\
\text{where } \begin{cases} 
\text{ip} = \sigma_0(\ell) \\
[p_0, \ldots, p_n] = \delta(\text{Id}_p, s, \text{par}) \\
[t_0, \ldots, t_n] = \delta(\text{Id}_p, s, \text{parype}) \\
\text{Statements} = \delta(\text{Id}_p, s, \text{stm})
\end{cases}
\]

4.6 Selections revisited

In Section 4.3 we presented the semantics of a Selection as an r-value. The semantics of a function call and new extensions to the function \( \mathcal{E} \) enable us now to present a generic definition for the semantics of a Selection as an r-value. We forget about the temporary extensions to \( \mathcal{E} \) for a Selection and use the following (definitive) extension instead.
\[ \mathcal{E}(\langle \text{Expression}, \text{Selectors} \rangle)(\delta)(\varphi)(\sigma) = d_i \]

where \[
\begin{align*}
  d &= \mathcal{E}(\text{Expression})(\delta)(\varphi)(\sigma) \\
  i &= \mathcal{T}(\text{Selectors})(\delta)(\varphi)(\sigma)
\end{align*}
\]

The complete class of Selection is covered with this definition. This class consists of the selections: \((\text{Shape, Selectors})\) and \((\text{Id, Selectors})\) and \((\text{FunctionCall, Selectors})\). This requires the function \(\mathcal{E}\) to be defined for \text{Shape, Id, and FunctionCall}; this was done in the preceding sections.

### 4.7 Flow of control

The 'flow of control' in a \(V\)-\(nus\) program is determined by \textit{While} terms, \textit{Iteration} terms, \textit{Forall} terms, \textit{Foreach} terms, \textit{If} terms, and \textit{Statements} (see Appendix A). The \textit{While} and \textit{Iteration} terms represent well-known loops. The first repeats executing a list of statements as long as a given condition remains valid. The latter defines an index space and executes a list of statements, parameterized with the index variables, for each element in the index space in a predefined order.

The \textit{Forall} term denotes a special kind of control statement, since it can be used to spawn parallel processes. In short: it describes a set of body instances that can be executed independently of each other. So, the body instances are not cumulatively executed, but all start in the same program state. It is therefore suited to serve as a statement that creates parallelism. Further details about the \textit{Forall} term are postponed until Chapter 5.

The \textit{Foreach} term looks like the \textit{Forall} term, but executes its body instances cumulatively. It differs from the \textit{Iteration} term in that the order of executing body instances does not matter.

The \textit{If} term is equivalent to the familiar if \ldots \texttt{then} \ldots \texttt{else} \ldots language construct.

The \textit{Statements} denote a list of \(V\)-\(nus\) statements that are executed sequentially. The sequential composition is expressed by the 'comma' that separates the individual statements.

#### 4.7.1 Sequential composition

The sequential composition of \(V\)-\(nus\) statements is defined by function composition of the meaning function \(\mathcal{M}\). The declaration state \((\Delta)\) and the index state \((\Phi)\) that are used for a statement are always the same as those of its immediate predecessor. This is trivial for the declaration state since it is set only once during 'start-up' (see Section 4.2). The index state can only be altered \textit{within} a control statement. Once it is altered, the index state is passed to the \textit{Statements} of that control statement. It is therefore the meaning function for this control statement that may cause changes to the index state. When composing statements sequentially they all get the same index state, by definition.
CHAPTER 4. THE DENOTATIONAL SEMANTICS

Of course, the program state can be changed, and then this changed program state is passed to the subsequent statement. A formalization of the above description of sequential composition is presented with the following (additional) definition of the meaning function $\mathcal{M}$. Let $c_1$ and $c_2$ be two arbitrary $V$-nus statements (or lists of $V$-nus statements).

$$\mathcal{M}(c_1,c_2)(\delta)(\varphi) = \mathcal{M}(c_2)(\delta)(\varphi) \circ \mathcal{M}(c_1)(\delta)(\varphi)$$

Note that the comma in $\mathcal{M}(c_1,c_2)$ is part of the $V$-nus language, and does not separate arguments of $\mathcal{M}$.

As was already mentioned in Chapter 2, a block of statements (Statements) does not (yet) denote a new scope. Therefore, the meaning of statements [actions] is equal to the meaning of the list actions itself. So, we extend $\mathcal{M}$ with the simple definition:

$$\mathcal{M}(\text{statements} \ [\cdot]) = \mathcal{M}(\cdot)$$

The constructor ‘statements’ is present in the language such that the compiler can easily parse a $V$-nus program without any shift-reduce conflicts.

4.7.2 Conditional flow

We start with the straightforward description of the conditional jump (CJ): the If term.

**Definition 4.7.1** The function $CJ : \mathcal{L}(If) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma$ is defined as

$$CJ(If)(\delta)(\varphi)(\sigma) = \begin{cases} \mathcal{M}(\text{Statements}_1)(\delta)(\varphi)(\sigma) & \text{if } \mathcal{B}(\text{Expression})(\delta)(\varphi)(\sigma) \\ \mathcal{M}(\text{Statements}_2)(\delta)(\varphi)(\sigma) & \text{otherwise} \end{cases}$$

where If = if Expression Statements$_1$, Statements$_2$

The meaning of an If term is completely defined by the function CJ. So, the function $\mathcal{M}$ is extended with

$$\mathcal{M}(\text{If}) = CJ(If)$$

The While term cannot be defined recursively in a straightforward matter. In a first attempt one would like to define the While semantics as

$$\mathcal{M}(\text{While})(\delta)(\varphi)(\sigma) = \begin{cases} \mathcal{M}(\text{While})(\delta)(\varphi)(\sigma') & \text{if } \mathcal{B}(\text{Expression})(\delta)(\varphi)(\sigma) \\ \sigma & \text{otherwise} \end{cases}$$

where \( \sigma' = \mathcal{M}(\text{Statements})(\delta)(\varphi)(\sigma) \)

\( \text{While} = \text{while Expression Statements} \)
4.7. FLOW OF CONTROL

But this approach violates the principle of compositionality which states that the meaning of any syntactic phrase may be defined only in terms of the meanings of its proper subparts [88]. This problem has been solved for years, so we use the standard solution and refer to [6] and [88] for a more detailed explanation.

We introduce the functions $CL_i$ for a conditional loop as to count down (from $i$) the number of body instances of the While term. These functions have the While term as input, and cumulatively 'execute' the body in the given states.

**Definition 4.7.2** The functions $CL_i : B(Statements) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma$ are defined for all $i \in \mathbb{N}$ as follows.

\[
CL_0(\text{While})(\delta)(\varphi)(\sigma) = \perp \\
CL_{i+1}(\text{While})(\delta)(\varphi)(\sigma) = \begin{cases} 
CL_i(\text{While})(\delta)(\varphi)(\sigma') & \text{if } B(\text{Expression})(\delta)(\varphi)(\sigma) \\
\sigma & \text{otherwise}
\end{cases}
\]

where \[
\begin{align*}
\sigma' &= M(\text{Statements})(\delta)(\varphi)(\sigma) \\
\text{While} &= \text{while Expression Statements}
\end{align*}
\]

The meaning of the While term is then defined in terms of the least upper bound of the functions $CL_i$. The meaning function $M$ is now extended with the definition:

\[
M(\text{While}) = \text{lub}_{i\in\mathbb{N}} CL_i(\text{While})
\]

A proof that this lub exists is omitted, since it can be given by standard techniques used in denotational semantics. Such a proof is based on the property that $(\Sigma, \sqsubseteq)$ is a cpo, which is shown in Section 3.3.

4.7.3 Iterated flow

For the semantic definition of a sequential loop (SL) we will use a special order on the index space of this loop. Then, the meaning of a sequential loop is as follows: start with the meaning of the body instance of the 'smallest' index. The program state after this execution is the context for the 'next' body instance. The final program state is the state after all body instances have been executed, one after each other.

In order to make this precise, we first introduce a lexicographical order on bounded sets. With the lexicographical order a so called $\Phi$-order is defined on index states. Finally, it is shown that a notion of an index state minimum exists. This minimum is used to indicate the first index state in which a sequential loop will start (and recursively, it indicates the next index state too).
Mathematical intermezzo

We follow the definition of a lexicographical order as it is presented in [102].

Definition 4.7.3 Let \( a = (a_0, \ldots, a_n) \) and \( b = (b_0, \ldots, b_n) \) be two arbitrary vectors. Let \( bs \in BSS \) such that \( a, b \in bs \). Let \( x \in \{0, \ldots, n\} \). The relation \( a <_x b \) is defined as

\[
a <_x b \iff \forall i \in \{0, \ldots, x-1\}, \text{proj}_i(a) = \text{proj}_i(b) \land \text{proj}_x(a) < \text{proj}_x(b)
\]

The lexicographical order is then defined by the relation \( a <_\alpha b \leq bs \times bs \) in the following way.

\[
a <_\alpha b \iff \exists x \in \{0, \ldots, n\}: a <_x b
\]

The following lemma is presented which is used in a lemma for the \( \Phi \)-order. It is needed to prove that a minimum on index states exists. A proof of the following lemma is omitted since it can be found in [102] and can easily be verified.

Lemma 4.7.1 Let \( a = (a_0, \ldots, a_n) \), \( b = (b_0, \ldots, b_n) \), \( c = (c_0, \ldots, c_n) \) be arbitrary vectors and let \( bs \subset BSS \) such that \( a, b, c \in bs \).

1. For every \( x \in \{0, \ldots, n\} \): \( a <_x b \) is an irreflexive partial order on \( bs \).

2. Either \( a = b \), or a unique \( x \in \{0, \ldots, n\} \) exists such that either \( a <_x b \) or \( b <_x a \).

3. For every \( x, x' \in \{0, \ldots, n\} \): \( a <_x b \land b <_{x'} c \Rightarrow a <_{x''} c \) where \( x'' = \min(x, x') \).

The lexicographical order \( (=_\alpha) \) allows ordering vectors of bounded sets. An ordered representation of an index state (an ordered index state, for short) represents such a vector. For instance, an index state \( \varphi \) for which applies \( \varphi(i_1) = n_1 \) and \( \varphi(i_2) = n_2 \) can be ordered as \( ((i_1, n_1), (i_2, n_2)) \) (clearly, it is not the only ordering). The vector it represents now is: \( (n_1, n_2) \). The \( \Phi \)-order (defined below) allows that an ordered index state \( I_1 \) is 'smaller' than an ordered index state \( I_2 \) if and only if the represented vector of \( I_1 \) is lexicographically smaller than the represented vector of \( I_2 \). This is defined in the following definition.

Definition 4.7.4 Let \( T_k = \{ t \mid t \text{ is an ordered representation of an arbitrary } \varphi, \ |\varphi| = k \} \) be the set of ordered index states of length \( k \). The relation \( =^2_\Phi \subset T_k \times T_k \) is defined, for all \( k \in \mathbb{N} \), as

\[
((i_0, n_0), \ldots, (i_{k-1}, n_{k-1})) =^2_\Phi ((j_0, m_0), \ldots, (j_{k-1}, n_{k-1})) \iff (n_0, \ldots, n_{k-1}) = (m_0, \ldots, m_{k-1})
\]
4.7. FLOW OF CONTROL

The relation \(<_\Phi \subseteq \mathcal{P}(T_k) \times \mathcal{P}(T_k)\) is then defined for the equivalence classes of \(=^2_\Phi\) as

\[
\left[\left(\langle i_0, n_0 \rangle, \ldots, \langle i_{k-1}, n_{k-1} \rangle\right)\right] =^2_\Phi \left[\left(\langle j_0, m_0 \rangle, \ldots, \langle j_{k-1}, m_{k-1} \rangle\right)\right] \leftrightarrow \left(\langle n_0, \ldots, n_{k-1} \rangle \preceq_\alpha \langle m_0, \ldots, m_{k-1} \rangle\right)\]

The relation \(\leq_\Phi\) is defined as \([x] \leq_\Phi [y] \iff (x =^2_\Phi y \land [x] <_\Phi [y])\) as usual.

The definition of \(\leq_\Phi\) is well-defined, since it makes no difference which representative of an equivalence class is used to represent this class. From now on, we will use the notation \(x \leq_\Phi y\) when \([x] \leq_\Phi [y]\) holds. We have the following lemma for the \(\Phi\)-order.

**Lemma 4.7.2** For all sets \(T_k\) with \(k \in \mathbb{N}\), where \(T_k = \{t \mid t\) is an ordered representation of an arbitrary \(\Phi, |\Phi| = k\}:

1. \(<_\Phi\) is an irreflexive order on \(T_k\).
2. \(<_\Phi\) is a transitive order on \(T_k\).
3. \(<_\Phi\) is a complete order on \(T_k\).

**Proof** 1 and 2 are easy to deduce with the aid of Lemma 4.7.1. 3 can be proved by using that \(<\) is a complete order on \(\mathbb{N}\), and thus \(<_\alpha\) is a complete order on \(bs, bs \in \mathcal{B}\).

One more definition is needed before we can prove that one minimum exists for a special set of ordered index states. This definition is used to describe this ‘special set’. That is, we are specifically interested in index states that are defined for the same set of index variables. In case we have to deal with an iteration term, each body instance will get an index state that is defined for the same index variables as the index states of the other body instances. For this reason we define a domain equivalence relation: two ordered index states are ‘domain equivalent’ when they contain the same index variables at the same place.

**Definition 4.7.5** Let \(T_k = \{t \mid t\) is an ordered representation of an arbitrary \(\Phi, |\Phi| = k\}\) for an arbitrary \(k \in \mathbb{N}\) be the set of ordered index states of length \(k\). Let \(t = (t_0, \ldots, t_{k-1}), t' = (t'_0, \ldots, t'_{k-1}) \in T_k\) be two ordered index states. The domain equivalence relation \(=^1_\Phi \subseteq T_k \times T_k\) is defined as

\[
t \dashv^1_\Phi t' \iff \forall i \in \{0, \ldots, k - 1\}. \text{proj}_0(t_i) = \text{proj}_0(t'_i)\]

\(t\) and \(t'\) are domain equivalent.
Now, enough mathematical building blocks have been created to claim our conclusion. With the relation \( \leq_\Phi \) on ordered index states that are domain equivalent we pose that exactly one minimum exists in the set of equivalence classes of these ordered index states.

**Theorem 4.2** Let \( \varphi_0, \ldots, \varphi_k \in \Phi \), such that for all \( 0 \leq i, j \leq k \). \(|\varphi_i| = |\varphi_j|\). Let \( t_p \) be an ordered representation of \( \varphi_p \) for \( 0 \leq p \leq k \) such that for all \( 0 \leq i, j \leq k \). \( t_i =_\Phi t_j \). Then

\[
\forall i \in \{0, \ldots, k\} \exists! x \in \{0, \ldots, k\}. t_x \leq_\Phi t_i
\]

We say that \( t_x = \min_\Phi(\{t_0, \ldots, t_k\}) \).

**Proof** We abbreviate the set \( \{0, \ldots, k\} \) to \( N \).

(1) First we show that at least one minimum exists.

Suppose there is no minimum. Then we have

\[
\forall i \in N \lnot \forall x \in N. t_i \leq_\Phi t_x
\]

So

\[
\forall i \in N \exists x \in N. t_i \not\leq_\Phi t_x
\]

(Lemma 4.7.2)

\[
\forall i \in N \exists x \in N. t_x \leq_\Phi t_i
\]

Contradiction; the set \( \{t_i | i \in N\} \) is a bounded set. So there is at least one minimum.

(2) Secondly, we show there is at most one minimum.

Suppose two minima \( t_x \) and \( t_y \) exist. Then we yield

\[
\forall i \in N. t_x \leq_\Phi t_i \text{ and } \forall i \in N. t_y \leq_\Phi t_i
\]

So

\[
t_x \leq_\Phi t_y \text{ and } t_y \leq_\Phi t_x
\]

(Lemma 4.7.2)

\[
(t_x =_\Phi t_y)
\]

So, exactly one minimum exists.

\[\square\]

**Back to semantics**

With the mathematical intermezzo it is shown that index states can be ordered. This is used to specify the semantics of a sequential loop. The syntax of such a loop is iteration Cardinalities Statements. The Cardinalities define the index space of the loop (described by the domain propagation function \( DP \) of Definition 4.4.3). The minimum of this index space is the ordered index state in which the loop starts; i.e. it defines the first body instance. The \( \leq_\Phi \) order defines the subsequent body instances. The semantics of a sequential loop are defined in terms of the function \( SL \).

**Definition 4.7.6** Let \( T_k = \{t | t \text{ is an ordered representation of an arbitrary } \varphi, \ |\varphi| = k \} \) for an arbitrary \( k \in \mathbb{N} \) be the set of ordered index states of length \( k \). The function \( SL : \mathcal{L}(\text{Statements}) \rightarrow \mathcal{P}(T_k) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma \) is defined as follows.


\[ SL(\text{Statements})(T)(\delta)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } T = \emptyset \\ SL(\text{Statements})(T \setminus \{t\})(\delta)(\varphi)(\sigma') & \text{otherwise} \end{cases} \]

where \[
\begin{align*}
t &= \min_\delta(T) \\
\varphi' &= \varphi \circ (t) \\
\sigma' &= M(\text{Statements})(\delta)(\varphi')(\sigma)
\end{align*}
\]

The set \( T \) represents a set of ordered index states. The ordered index state \( t \) is the minimum of \( T \). In the next recursive call \( t \) is removed from \( T \) such that taking the minimum each time defines the loop execution order. Note that the body instances are aware of the program state changes made in the preceding body instances. The changed index state \( \varphi' \) is of no use in the subsequent body instance. For each body instance a fresh index state is created from the index state of the outer loop and the minimum index state of the current loop. The meaning of a sequential loop, an \textit{Iteration}, is given by the following extension to the meaning function \( M \).

\[ M(\text{Iteration})(\delta)(\varphi)(\sigma) = SL(\text{Statements})(T)(\delta)(\varphi)(\sigma) \]

where \[
\begin{align*}
\text{Iteration} &= \text{iteration} \\
T &= DP(\text{Cardinalities})(\delta)(\varphi)(\sigma)
\end{align*}
\]

In the next chapter one will find the semantics for the terms \textit{Forall}, \textit{Foreach}, and \textit{Reduce}, where the mathematical intermezzo is used again.
Chapter 5

Implicit parallelism

...the theoretical physicist's picture of the world
...demands the highest possible standard of rigorous precision in the description of relations, such as only the use of mathematical language can give.
...Supreme purity, clarity, and certainty at the cost of completeness.

Principles of Research (Motiv des Forschens)
Albert Einstein

5.1 Introduction

We distinguish 'implicit parallelism' and 'explicit parallelism' in V-nus. Explicit parallelism is denoted by V-nus terms that explicitly require more than one process, for instance Send and Receive\(^1\). Implicit parallelism is harder to describe in a few words; in fact, a complete V-nus program might express implicit parallelism. An implicitly parallel V-nus term is a term for which the compiler tries to detect whether it can generate code for parallelism. It can always generate code for sequential execution, since the semantics of these V-nus terms do not depend on the existence of more than one process. Ben-Ari [11] calls this kind of parallelism 'concurrency', and describes it with "The word 'concurrent' is used to describe processes that have the potential for parallel execution".

In this chapter we focus on the implicitly parallel V-nus terms Foreall, Foreach, and Reduce. The main focus is on the Foreall term. With this term the programmer can specify large/complex computations that are recognized by the compiler to generate code for parallelism. Although the Foreach term has sequential semantics, it is an efficient program construct to use in a parallel context. The Reduce term is a well known language construct in data parallel languages; it represents a reduction computation, but leaves it to the compiler how to do this computation. A parallel computation is often suitable.

\(^1\)This description is not quite true, since a process can send to itself and has no need for a different process, but we refer to Chapter 1 and Chapter 6 for a more detailed explanation.
Reduce-like constructions in functional programming languages often have sequential semantics. In our semantics a set of computation traces is specified; our language \( V-nus \) can pass the \( \text{Reduce} \) term to the underlying communication mechanism which might have a single construct for this operation (for instance, a \text{SUM}). It would then be a waste of computation resources if this \( \text{Reduce} \) term was first transformed to other language terms, where these other terms would later be transformed (if possible) to this single construct of the communication mechanism.

Together with a reduce operation a scan operation is often present. Such a scan operation can be seen as a generalization of the reduce operation. \( V-nus \) does not have a special language construct for this scan operation, but we nevertheless spend some words on this operation and show how it can be represented in \( V-nus \).

5.1.1 Notational conventions

Since this chapter covers language constructs of different languages with a similar meaning, we introduce denotations that refer to any of these constructs rather than one in particular. We will use the word \text{iteration} to mean any generic index-driven iteration language construct. Such a construct consists of two parts: an \text{index-space specification} and a \text{body}. In \( V-nus \) these are called \text{Cardinalities} and \text{Statements} respectively. The body is parameterized with respect to, and will be executed for, every index in the index-space specification. Each separate instance of the body is called a \text{body instance}. We use the word \text{forall} to represent a specific kind of \text{iteration} that allows a concurrent execution of the body-instances. Furthermore, we sometimes need to refer to the domain of a function. In general, for an arbitrary function \( f : X \rightarrow Y \) we use the notation \( \text{dom}(f) \) to refer to \( X \).

5.1.2 The \text{forall} statements

The \text{forall} statement is an important language construct in many (data) parallel languages \cite{Boo, Con, For, HPF, For, HPF}. It specifies which computations can be performed independently. Although its necessity is widely accepted, the \text{forall} definition differs per language. The \text{forall} statement in each of the languages was designed with specific implementation criteria in mind.

We think it is important to have a clear and generalized semantics for \text{forall} statements in all languages in which they occur. This chapter defines a generalized \text{forall} statement and discusses its semantics and implementation. We will show how \text{forall} constructs as found in the languages \text{Booster} \cite{Boo}, \text{Connection Machine Fortran} (CM Fortran) \cite{Con}, and \text{High Performance Fortran} (HPF) \cite{HPF} are mapped to this generalized \text{forall} statement without forfeiting semantics and efficiency. Furthermore, the \text{forall} statement we define has the ability to spawn more complex independent activities than can be found in these languages. Having a single language construct that spawns a parallel loop improves the orthogonality of a language. It is our opinion that this \text{forall} statement is not only suited to
an intermediate representation, but can also be adopted at the syntactic level in high-level parallel languages.

It is our goal to find a `forall` statement that complies with the following requirements:

- The denotational semantics of a `forall` statement must represent only one possible program state change; that is, only one outcome should be possible after execution of a `forall`.

- It must be possible to implement the `forall` statement efficiently. This means that the administration that is needed to execute the `forall` should not use excessive amounts of computational resources.

- The `forall` statement must be capable of representing a wide class of `forall` definitions as can be found in (data) parallel languages.

- It must be possible to give a concise operational semantics of the `forall` statement that can easily be understood.

### 5.1.3 Chapter overview

Section 5.2 presents an overview of different types of iteration statements, one of which is chosen for the V-nus language. In Section 5.3 we investigate the `forall` statement in different (data) parallel languages. The meaning of the `forall` statement of V-nus is defined in detail in Section 5.4. In Section 5.5 we explain how the V-nus `Forall` can be implemented efficiently. Subsequently, it is shown in Section 5.6 how different interpretations of a `forall` statement can be represented in V-nus. Sections 5.7 and 5.8 present the semantics of the V-nus language constructs `Foreach` and `Reduce` respectively. In Section 5.9 we again consider the goals of this chapter, and we draw our conclusions.

### 5.2 Different types of iteration

In the set of iteration statements, we can identify two extremes: the sequential loop and the completely parallel loop. Below, we present a short stock-taking within the different types of iteration. It is not meant as a complete list of these types, but gives some background for the implicit parallel loops as these are constructed for V-nus.

**The sequential iteration** An example of the sequential iteration is the conventional for-loop in V-nus it is defined as the `Iteration` term (the non-terminal `Iteration!`). The body-instances are executed one after another, in some sequential order. Loop carried data dependencies can exist (i.e. data dependencies between different body instances). See Chapter 4 for a precise description of the `Iteration` term.

**The chaos iteration** The body-instances are executed completely concurrently. All body-instances work on the same memory locations, and no assumptions are made about
the order in which writes to and reads from these variables take place. A non-
deterministic behaviour can be a result of this model of execution.

Besides these extremes we present a number of other iteration statements below.

The merge iteration The body-instances are executed completely concurrently. All
body-instances work on their own copy of the program state, so determinism is guar-
anteed. At the end of the iteration statement all the now-changed individual program
states of the body-instances must be merged back into a single parent program state
by a merge function.

The statement-atomic iteration The body-instances are executed concurrently, but
the statements within the body are considered to be atomic. This means that during
the execution of a statement $S$ it is guaranteed that no other body-instances will be
updating the value of any of the variables used in $S$.

The body-atomic iteration The body-instances are executed concurrently, but the en-
tire body is considered to be atomic; i.e. during the execution of a body-instance $i$ it
is guaranteed that no other body-instances will be updating the value of any of the
variables used in body-instance $i$.

These intermediate forms of iteration statements are called forall statements. Both
the statement-atomic and the body-atomic forall statement imply a certain amount of
synchronization and variable-shielding. We have chosen the merge forall semantics for the
V-nus Forall term, because it has the most potential parallelism (given the requirements
for the forall concept we are looking for) and is well-suited for use in programming. The V-
nus Foreach term is another example of the sequential iteration. The Foreach term allows
for a successive execution of the body instances. However, the order of body instance
executions is not predefined; any order can be chosen.

Note the main difference between a Forall term and a Foreach term: in the first, the
body instances all start executing in (a copy of) the same program state; in the latter, a
body instance starts in the final program state of its predecessor.

5.3 Related work

Both data parallel languages and control parallel languages use the concept of a forall
statement to denote the spawning of concurrent actions. There is a common trade off in
the definitions of forall statements in these languages: constraints on the body decrease
the potential parallelism, but lack of these constraints may cause non-determinism. An as-
signment in a specific body-instance may affect the computation of another body-instance,
when these body-instances share the same variable. The outcome of a forall statement
is then dependent on the order of computation. In general, it is impossible to know at
compile time which data elements are assigned to. The solution for this problem is putting
5.3. RELATED WORK

restrictions to forall statements to reduce undesirable behaviour. Function calls and procedure calls complicate the task of finding well-defined restrictions even more, since it is hard to analyze their effect on the program context in general.

One of the first versions of the forall statement was introduced by Thinking Machines Corporation in CM Fortran [26]. It is used to distribute computations over the processing elements of the Connection Machine (CM). The keyword forall indicates that the body-instances can be executed independently. The body-instances consist of one assignment with a left-hand side that is not assigned to by another body-instance. The use of certain kinds of expressions, such as user defined functions and assignments to array sections that depend on the index variable, always causes the forall statement to be executed serially.

Vienna Fortran [101] defines a broader forall statement by permitting private variables. These variables are known only in the forall statement in which they are declared, and each body-instance has a separate copy. A body-instance can consist of any legal FORTRAN executable statement. Tightly nested forall statements can be used to specify multiple levels of parallelism. Vienna Fortran also restricts the forall body by requiring that a value written in one body-instance is neither read (define-use dependency) nor written (define-define dependency) in any other body-instance (see [102] for a description of define-use and define-define dependencies). The result is always deterministic.

Experiences with the forall statement in the Fortran dialects CM Fortran, Vienna Fortran, and Fortran D [42] led to the construction of the HPF forall. CM Fortran uses the forall statement to create parallelism explicitly by distributing body-instances over the CM. Vienna Fortran uses the forall statement to indicate that the different body-instances are independent and can be logically executed in parallel. In HPF [41] it is the distribution of data that introduces parallelism.

The HPF forall statement consists of a single assignment statement. The left-hand side of each body-instance of this assignment can only be assigned once. This excludes define-define dependencies. Execution of the forall statement requires the right-hand sides of the body-instances to be evaluated before these are assigned to the left-hand sides. This implies that a synchronization is needed. Only function calls to pure functions (functions that have no side effect) may be used in the right-hand side. It is then assured that define-use dependencies leave the outcome of the forall statement deterministic.

It is allowed to have multiple statements in the HPF forall body\(^2\), but this means that each assignment of the body is executed for the complete index space; i.e. as if the assignments were written as separate forall statements in the same order (see Example 5.3.1 and Section 5.6).

---

\(^2\)HPF distinguishes between forall statements and forall constructs; the latter may have multiple statements in their bodies.
Example 5.3.1 The forall construct of HPF.
Consider the following HPF forall construct, where \( i \) ranges from \( a \) to \( b \), and where Assignment1 and Assignment2 are valid assignments that may occur in a forall construct.

\[
\text{FORALL (i=\text{a:b})}
\begin{align*}
\text{Assignment1} \\
\text{Assignment2}
\end{align*}
\text{END FORALL}
\]

This forall construct can be rewritten to the following code fragment, which has the same meaning as the code above.

\[
\text{tempa = a} \\
\text{tempb = b} \\
\text{FORALL (i=\text{tempa : tempb}) Assignment1} \\
\text{FORALL (i=\text{tempa : tempb}) Assignment2}
\]

In addition a directive independent has been introduced for both \textbf{do} loops and forall statements. The directive assures the compiler that the body-instances can be executed in an arbitrary order, without any computational differences in the result. In case of the multiple statement forall this means no synchronization is needed between the statements. Both the single assignment and the multiple assignment forall statement of HPF are used in the same form with the same semantics in Fortran 95, according to the proposed revision \[40\].

The data parallel language Booster \[18\] has no forall keyword. It is possible to assign array sections in parallel by using an aggregate assignment. Unambiguous semantics are enforced by the requirement that no element is used as a target before it is used as a source. Function calls do not make such analysis harder, since Booster requires the functions to be referentially transparent; i.e. no side effects occur and no global variables are accessed.

In the control parallel language SuperPascal \[51\] the forall statement is used to denote an array of parallel processes. A severe restriction is imposed on the forall body to prevent ambiguous computations: the body may not assign to a variable. This implies that a body-instance must output its results through a communication channel or a file. Otherwise, the results will be lost when the body-instances terminate. Procedure calls can be used in the body, which causes no problems under the given circumstances.

The forall statement in Compositional C++ \[20\], denoted by the keyword parfor, also initiates the parallel execution of the body-instances. Multiple statements are allowed in the forall body, where the statements of a specific body-instance are executed sequentially. Note that this is in contrast with the multiple statement forall of HPF. No copies are
made of data that is used in the body-instances, so loop carried dependencies can lead to
deterministic results.

The Myrias parallel do uses copy-in/copy-out semantics [10]. When a program executes
a parallel do construct, parallel tasks are created, one for each iteration of the parallel do.
Each task gets a separate copy of the parent program state. At the end of the parallel do
all child program states are merged to form the new program state. It is, however, not
explained how this merging can be done efficiently.

Li and Wolfe [65] mentioned the difficulties in defining well-behaved parallel constructs
without making arbitrary decisions. They developed a framework for analyzing the behav-
ior and relations of various sequential and parallel control constructs. Their DoPar
iteration has a similar meaning as the merge forall, described in Section 5.2 and is based
on the parallel do of the Myrias system. Also there, it is not mentioned how to implement
this general iteration construct efficiently. Using their framework they present how and
when different loop constructs can be substituted by another loop construct.

The Foreach term of V-nus is similar to the DoAny loop construct described in [100].
Both loop constructs allow an arbitrary loop order, but parallel execution of two data
dependent statements may be disallowed.

In the remainder of this chapter we will use the forall statements of Booster, CM
Fortran, and HPF as representatives of the many forall definitions that can be found in
literature on data parallel languages.

## 5.4 The semantics of Forall

Similar to the other languages, the V-nus Forall statement is represented by the syntax:
forall Cardinalities Statements. The term Cardinalities specifies the range of the index
variable; the term Statements represents the list of statements that will be executed for
each value of the index variable (see Example 5.4.1).

---

### Example 5.4.1 The V-nus Forall statement

Consider: forall [(i,3)] statements [assignment (a, [ ] i)]. The index variable is i and ranges
over 0, 1 and 2. The body is statements [assignment (a, [ ] i)], an example of a body instance
is statements [assignment (a, [ ] 1)].

---

Body-instances of the V-nus forall statement are to be executed completely independ-
ently. By this we mean that data that can be changed by a body-instance i will not
affect the computation of another body-instance j. However, a global interference is still
possible when there is a define-define dependency between the possible body-instances; i.e.
two body-instances that write to the same variable. We introduce the term ‘deterministic
forall statement’.
Definition 5.4.1 A ‘deterministic forall statement’ is a forall statement where no define-dependency is present between any two different body instances of the forall statement.

We want to record the concept of the forall statement in our semantic model, such that we can use this model to reason about a program that uses this implicit form of parallelism. The Forall term expresses a set of independent computations. These computations can be done in parallel or sequentially. Therefore, the type of the program state (Σ) remains the same; there is no need for a type change to define the meaning of the terms Forall and Foreach.

The semantics of a program fragment can be given by a program state change, represented by a pair \((σ, σ')\) of program states. In case of the forall statement, program state changes are computed for all body-instances. Say, for body-instance \(i\) the state change \((σ, σ_i)\) is computed. Then the different program states \(σ_i\) (for all \(i\)) are merged into the final program state \(σ'\), which will be the program state after the forall statement has been executed. This merge operation consists of two actions. First \(σ_i\) is compared with \(σ\), providing only the difference \(diff_i\) between these program states. Secondly, all elements of \(diff_i\) will be put into \(σ'\). This is done for all \(σ_i\) in arbitrary order.

For the definition of the semantics of the Forall term there is no need to order the index space (denoted by the Cardinalities) as was done for the Iteration term (see Section 4.7). As was mentioned above, we need a difference operation and a merge operation to specify the semantics of the Forall. The merge operation can already be expressed with the known replacement function \(σ\) of Definition 3.6.1. For the difference operation the operator ‘\(\div\)’ is introduced. To avoid cumbersome notation when using subscripts we use this new operator for the well-known difference on sets.

Definition 5.4.2 Let \(f, g : X \rightarrow Y\) be functions on arbitrary sets \(X\) and \(Y\). Let \(Z = \mathcal{P}((X \times Y))\). The difference function \(\div : Z \times Z \rightarrow Z\) is defined such that

\[
f \div g = f \setminus g
\]

where \(\div\) is left-associative.

So, the expression \(f \div g\) means: “remove all pairs of \(g\) from \(f\)” and the so changed \(f\) represents the difference. In the semantic model of \(V\)-nus it is not necessary to collect the pairs of \(g\) that do not exist in \(f\) as parts of the difference. The reason for this is that program states always ‘grow’. More precisely: for any program \(c_1, c_2\), where \(c_1\) and \(c_2\) are arbitrary program fragments, we have that \(M(c_1)(δ)(ϕ)(σ)\) is included in \(M(c_1, c_2)(δ)(ϕ)(σ)\). This ‘included’ relation is defined as ‘the substates of the first operand being a subset of the corresponding substates of the second operand’. In other words, there is no \(V\)-nus term that causes a semantic function to remove something from the program state.

The difference computation is used to see what extra changes have been made in the program state, compared to the program state that had been computed for a smaller part of the same program. That is, we are interested in differences between program states.
$\sigma_1$ and $\sigma_2$ where $\sigma_1$ is "included" in $\sigma_2$. The expression $\sigma_2 \triangleleft^* \sigma_1$ will return the wanted differences (the notation is explained shortly). This explains why $f \triangleleft q$ does not collect pairs of $g$ that do not reside in $f$. In Section 5.4.1 we formalize the idea of "growing" program states.

In order to express differences between program states, the following short hand notations are used (similar to those used for the replacement function).

**Notation:** For an arbitrary program state $\sigma = (\sigma_V, \sigma_L)$, the expressions: $\sigma \div V X$ and $\sigma \div L Y$ are shortcuts for: $(\sigma_V \div X, \sigma_L)$ and $(\sigma_V, \sigma_L \div Y)$ respectively. For another arbitrary program state $\sigma' = (\sigma'_V, \sigma'_L)$, the expression: $\sigma \triangleleft^* \sigma'$ is equal to $\sigma \div V \sigma'_V \div L \sigma'_L$.

Now we can present the semantic function $PL$ that defines the state change of a parallel loop.

**Definition 5.4.3** Let $T_k = \{ t | t \text{ is an ordered representation of an arbitrary } \varphi \in \Phi, |\varphi| = k \}$ for an arbitrary $k \in \mathbb{N}$ be the set of ordered index states of length $k$. Let $(j_0, \ldots, j_n)$ be a predefined permutation of $(0, \ldots, n)$. The function $PL : L(Statements) \rightarrow P(T_k) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma$ is defined as follows. 

\[
PL(Statements)(T)(\delta)(\varphi)(\sigma) = \sigma'
\]

where 

\[
\begin{align*}
T &= \{ t_0, \ldots, t_n \} \\
\forall i \in \{0, \ldots, n\}, \varphi_i &= \varphi \triangleleft t_i \\
\forall i \in \{0, \ldots, n\}, \sigma_i &= M(Statements)(\delta)(\varphi_i)(\sigma) \\
\forall i \in \{0, \ldots, n\}, \text{diff}_i &= \sigma_i \triangleleft^* \sigma \\
\sigma' &= \sigma \triangleleft^* \text{diff}_0 \triangleleft^* \ldots \triangleleft^* \text{diff}_n
\end{align*}
\]

To avoid a cumbersome notation the permutation of $(0, \ldots, n)$ is presented as being "predefined" rather than being an argument of the function $PL$.

The set $T$ represents a set of ordered index states. Instead of cumulatively computing program states in a predefined order (like the $\leq_{\Phi}$ order for the sequential loop), a set of program states $\sigma_t$ is computed that all use the original program state $\sigma$. The computed differences $\text{diff}_i$ represent a program state that only contains those changes that are made by the body instance that goes with the ordered index state $t_i$. The final program state $\sigma'$ is thus expanded with the newly made changes.

The meaning of the program construct **forall** Cardinalities Statements is given by the following definition of the meaning function $M$ (using the function $DP$ of Definition 4.4.3):

\[
M(\text{forall})(\delta)(\varphi)(\sigma) = PL(Statements)(T)(\delta)(\varphi)(\sigma)
\]

where 

\[
\begin{align*}
\text{forall} &= \text{forall} \text{ Cardinalities Statements} \\
T &= DP(\text{Cardinalities})(\delta)(\varphi)(\sigma)
\end{align*}
\]

When no two different body instances write to the same variable the order in which the differences are merged back to the original program state does not matter. Example 5.4.2 shows how to compute the meaning of a deterministic **forall** term.
Example 5.4.2  The meaning of a deterministic Forall.
Consider the simple Forall term: forall [(i,3)] statements [assignment (A,[]) i]. Assume that δ, φ, and σ are valid states to serve as a context in which this statement can be executed; and say, the current scope is s. The domain propagation of [(i,3)] is computed with: $DP([(i,3)])(δ)(φ)(σ) = \{(i, 0), (i, 1), (i, 2)\}$; a set of three ordered index states, which we abbreviate to $T$. The meaning of a parallel loop for these ordered index states is computed with: $PL(statements [assignment (A,[]) i])(T)(δ)(φ)(σ)$. This will compute intermediate program states $σ_i$ for $0 \leq i \leq 2$ for which apply: $σ_{V_i}(A, s)(i) = i$. So, the differences $diff$, each consist only of a pair $[((A, s), i), i]$ which are merged into the original program state $σ$. Thus, the call to $PL$ returns a program state $σ'$ for which holds: $σ'_{V}(A, s)(i) = i$ for $0 \leq i \leq 2$. The function $M$ calls $PL$ with the correct parameters.

5.4.1 Relating program states

Above we mentioned that when a program proceeds with its execution, the corresponding program state is growing. Here, we will formalize this assertion, and prove its correctness. First, a (partial) relation is introduced that defines whether one program state is included in another program state.

Definition 5.4.4  The relation $<_\Sigma$: $\Sigma \times \Sigma$ is partially defined as follows.

$$\sigma <_\Sigma \sigma' \iff dom(\sigma_V) \subset dom(\sigma'_V) \land dom(\sigma_L) \subset dom(\sigma'_L)$$

In a similar way, the relation $=_\Sigma$: $\Sigma \times \Sigma$ is partially defined as

$$\sigma =_\Sigma \sigma' \iff dom(\sigma_V) = dom(\sigma'_V) \land dom(\sigma_L) = dom(\sigma'_L)$$

As to be expected, the constraint $\sigma \leq_\Sigma \sigma'$, for arbitrary $\sigma, \sigma' \in \Sigma$, is a shortcut for $\sigma <_\Sigma \sigma' \lor \sigma =_\Sigma \sigma'$.

Usually, the domains of the location state ($\sigma_L$) are fixed, which leaves us with a complete relation on the view state ($\sigma_V$).

Without proof we state the following lemma.

Lemma 5.4.1  $\leq_V$ is a reflexive, transitive relation.

The proof is trivial, and based on the reflexivity and transitivity of $\subseteq$.

For the theorem below, we will use the following terminology:
5.4. THE SEMANTICS OF FORALL

- A singulary term is a term \( c_0 \in V\text{nus} \) that has no separator ‘,’ in its syntax that separates \textit{Actions}.
- The length of a term \( c \), notation \(|c|\), is the number of singulary terms in \( c \).

Note, that we often use the syntax: \( c_1,c_2 \) which represents \textit{one} \( V\text{-nus} \) term, since the comma ‘,’ is the \( V\text{-nus} \) separator and is part of the language.

We are now able to express the assertion about ‘growing’ program states formally with the following theorem.

\textbf{Theorem 5.1} Let \( \delta \in \Delta \) be a valid declaration state with respect to \( c \in V\text{nus} \). Let \( \varphi \in \Phi \) be an arbitrary index state. Then

\[
\forall c \in V\text{nus} . \forall \sigma \in \Sigma . \sigma \leq \Sigma M(c)(\delta)(\varphi)(\sigma)
\]

\(\Box\)

\textbf{Proof} The proof is based on induction on the length of \( c \).

1. It is easy to verify that for all singulary terms \( c_0 \in V\text{nus} \), we have: \( \sigma \leq \Sigma M(c_0)(\delta)(\varphi)(\sigma) \) (see the definition of \( M \) in Appendix B).

2. Assume that the induction hypothesis (I.H.) holds: \( \sigma \leq \Sigma M(c)(\delta)(\varphi)(\sigma) \) for all terms \( c \in V\text{nus} \), for which \(|c| = n\), for a certain \( n \in \mathbb{N} \). Consider another \( c' \in V\text{nus} \), \(|c'| \leq n\), such that \( c,c' \in V\text{nus} \). With compositionality of \( M \) we have:

\[
M(c,c')(\delta)(\varphi)(\sigma) = M(c')(\varphi)(\delta)(M(c)(\delta)(\varphi)(\sigma))
\]

According to the induction hypothesis (I.H.) we have:

\[
M(c)(\delta)(\varphi)(\sigma) \leq \Sigma M(c')(\delta)(\varphi)(M(c)(\delta)(\varphi)(\sigma))
\]

Summarizing, we have:

\[
\sigma \leq \Sigma M(c)(\delta)(\varphi)(\sigma) \leq \Sigma M(c,c')(\delta)(\varphi)(\sigma)
\]

With the transitivity of \( \leq \Sigma \) (Lemma 5.4.1) we conclude:

\[
\sigma \leq \Sigma M(c,c')(\delta)(\varphi)(\sigma)
\]

3. With 1. and 2. we have proven that \( \sigma \leq \Sigma M(c)(\delta)(\varphi)(\sigma) \) holds for all \( c \in V\text{nus} \) and all \( \sigma \in \Sigma \).

\(\Box\)
5.5 The implementation of \textit{Forall}

Implementing the \textit{Forall} term as presented in the previous section may cause problems when efficiency is considered. Merging the different program states of the body-instances is inefficient, since computing the difference between program states is time consuming.

Note that both in the context of compiler implementation and in the context of semantics a notion of program state is used. Although in both contexts the program state means 'the state of the program at a certain moment', it does not need to have the same 'syntax'. The program state used in the semantics is a representation of the program state used in the compiler. Since we spend some words on the implementation in this section, this difference is expressed in the notation of program states. The 'semantic program state' is still denoted by the Greek symbols: $\sigma, \sigma', \ldots$. The 'compiler program state' is denoted by the keywords: $ps, ps', qs, qs', \ldots$.

In order to arrive at an efficient implementation of the \textit{Forall} term, we take the following approach. At the start of a \textit{Forall} term the program state $ps$ is preserved. For the execution of a body-instance $i$ a subset $qs_i$ of $ps$ is used for the context in which this body-instance will be executed, where $i$ is an index of the index-space of the \textit{Forall}. Only the data that is needed in the body-instance is extracted from $ps$ and will be used for $qs_i$. Each time something needs to be read from memory, it is read from $qs_i$. When something needs to be written to memory, it is not only stored in $qs_i$, but the same store action is also performed on $ps$. In this way, each change that is made by a single body-instance is also visible in the global program state, but will not affect the other body-instances. This is how the final program state $ps'$ arises from the original program state $ps$, without the need for a merge or a difference operation (see Figure 5.1).

![Diagram](image)

Figure 5.1: Different strategies for computing the final program state after execution of a \textit{Forall}.
5.5. THE IMPLEMENTATION OF FORALL

5.5.1 Pragmas

The construction of $qs_i$ is dependent on the information the compiler has about the data that is used in the body-instance. This information can be generated automatically by well-known dependency analysis techniques and by hand via pragmas. A Pragma is an optional annotation for the compiler that gives additional information about a certain program construct. Pragmas that can be used for a Forall term specify which data should be copied in $qs_i$.

In general, Pragmas can precede any Action or Expression and inform the compiler on how to compile the preceded term. Pragmas are denoted with the keyword pragma followed by a list ofPragma-s (note the difference between the non-terminals Pragmas and Pragma). A Pragma comes in two forms, which can be seen as parameterized and non-parameterized pragmas. The latter has the syntax: flag $Id$, where $Id$ specifies the annotation. Examples of such annotations are:

readonly When a formal parameter is annotated with this pragma, the compiler may assume that this parameter is never assigned to.

location When a formal parameter is annotated with this pragma, the compiler assumes that this parameter is only passed location expressions. If the formal parameter is also annotated as readonly, the compiler can declare this as a 'reference parameter' in, for instance, C++. This can improve the performance of parameter passing for data structures of type shape.

The parameterized form of a Pragma is denoted as: value $Id$ Expression. Again, $Id$ expresses the kind of annotation, where Expression serves as the parameter for this annotation. Some examples are:

processors Shape When a program is annotated with this pragma, it will use the processor data structure specified by Shape. So, the pragma: processors shape[3,4] specifies a processor array of 3 by 4.

dependsOn Expressions This is a typical annotation for a Forall term. This is explained below.

5.5.2 Annotating the Forall term

If a Forall term is not annotated by a pragma, then the local program states $qs_i$ are created as explained earlier in this section. If a pragma is present the compiler relies on this information and only copies the given data structures for the accompanying program states $qs_i$. In our opinion, it is more useful to specify for which data structures a dependency exists, than it is to specify those structures for which no dependency exists. The syntax of a pragma for a Forall term is:

pragma [value dependsOn [Expression, ...]]
which expresses a dependency for the data structure(s) Expression. An empty list of
specifications (i.e. pragma [value dependsOn []]) means that no data needs to be copied.
Of course, it is the responsibility of the programmer to avoid the introduction of non-
determinism due to a pragma.

Especially when the compiler cannot determine what dependencies exist between the
body-instances, it is useful to be able to give additional information to the compiler. In
Example 5.5.1 is shown how the efficiency of a Forall term can be optimized by introducing
a pragma.

Example 5.5.1 Using pragmas for a Forall term.
Consider the program fragment:

```plaintext
forall [(i,n)] statements [ 
  assignment (A, [i]) (B, [(C, [i])]),
  assignment (B, [(C, [i])]) (A, [(i,+1)])
]
```

At compile-time it is unknown what elements of B are referenced. The conservative
approach is taken so that this Forall is characterized as non-deterministic. Furthermore, for
each body-instance qs_i a complete copy of B is created. If all elements of C are different
then each body-instance will write to a different element of B. In that case, there is no
need to create a copy of B in each qs_i. Note that for A it is necessary to create a local
copy. So we can safely annotate the Forall term as follows:

```plaintext
pragma [value dependsOn [(A, [(i,+1)])]] forall [(i,n)] statements [ 
  assignment (A, [i]) (B, [(C, [i])]),
  assignment (B, [(C, [i])]) (A, [(i,+1)])
]
```

which means that each body-instance qs_i must have a copy of (A, [(i,+1)]), and no other
copies are needed. When a pragma is used, it is assumed that the Forall is deterministic.

When using pragmas the execution model is slightly changed. Each time something
needs to be read from memory, it is read from qs_i if it exists in qs_i; otherwise it is read
from ps. Proper use of pragmas still guarantees determinism when the original program
was deterministic.

5.5.3 The implementation compared to the semantics
In the implementation of a deterministic Forall term, all differences between the program
states qs_i are collected in the global program state ps'. This is exactly as it is described
by the denotational semantics.

The denotational semantics use the same computation for both deterministic and non-deterministic Forall terms. That makes the result of a non-deterministic Forall term dependent on the computation order. In this case the efficient implementation of a Forall term may compute other results than the theory prescribes. In Example 5.5.2 a possible difference is presented between the computation used in the implementation, and the computation used to determine the semantics.

---

**Example 5.5.2** Difference between theory and implementation.

Consider the program fragment:

```
forall [(i,2)] statements [  
    assignment (a, []) i,  
    assignment (b, []) i  
]
```

The denotational semantics predict that the body-instance for \( i = 0 \) will result in the program state \( \sigma_0 \) such that \( \sigma_{0_i} \circ \sigma_0(a, s)((i)) = 0 \) and \( \sigma_{0_e} \circ \sigma_0(b, s)((i)) = 0 \). The body-instance for \( i = 1 \) will result in the program state \( \sigma_1 \) where \( a \) and \( b \) represent the value 1. \( \sigma' \) will then be either \( \sigma_0 \) or \( \sigma_1 \).

The implementation, on the other hand, may cause the following execution orders:

```
assignment (a, []) 0, assignment (b, []) 0, assignment (a, []) 1, assignment (b, []) 1 or  
assignment (a, []) 1, assignment (b, []) 1, assignment (a, []) 0, assignment (b, []) 0 or  
assignment (a, []) 0, assignment (a, []) 1, assignment (b, []) 0, assignment (b, []) 1 or  
assignment (a, []) 1, assignment (a, []) 0, assignment (b, []) 1, assignment (b, []) 0 or  
assignment (a, []) 0, assignment (a, []) 1, assignment (b, []) 1, assignment (b, []) 0 or  
assignment (a, []) 1, assignment (a, []) 0, assignment (b, []) 0, assignment (b, []) 1
```

which will lead to the same possible program states as predicted by the theory, plus the program state where \( a \) represents the value 0 and at the same time \( b \) represents 1, and the program state where \( a \) represents 1 and \( b \) represents 0. See also Figure 5.2.

---

In Example 5.5.2 both the body-instances write to the variables \( a \) and \( b \), which make the Forall statement non-deterministic. Theory and implementation only differ for non-deterministic Forall terms. We want to use a semantic model in which the outcome of a program (fragment) is unambiguous. When non-determinism is forced by a non-deterministic Forall term it is sufficient to mention that the outcome is unpredictable. There is no need for a semantic function that defines the set of all possible outcomes, since we are interested in a deterministic Forall only.
5.6 The *Forall* compared to other languages

As shown in Section 5.3, many languages have a notation that describes some independent iteration over an index space. However, the semantics of these constructs differ for each language. In this section, we compare the *forall* terms of the data parallel languages *Booster*, CM Fortran, and HPF, and we show how these differently defined *forall* terms can be mapped to the *V-nus* *Forall* term. The *forall* terms of these languages (except *V-nus*) are denoted in a syntax from a pseudo language. Of course, the so denoted terms have a one-to-one mapping to their original syntax.

CM Fortran as well as HPF use the same method for the evaluation of *forall* *IndexSpace Body*: first, evaluate the expressions in *IndexSpace*, then, evaluate all expressions present in *Body*, and finally, perform the assignments of *Body*. More detailed descriptions are given in the appropriate language specifications. Example 5.6.1 shows three essential occurrences of a *forall* term. We will interpret these *forall* terms in the four languages (if possible). More examples of *forall* statements can be found in Appendix C, where they are interpreted in the languages CM Fortran, HPF, and *V-nus*. 

Figure 5.2: Computing the final program state after executing the assignments of Example 5.5.2. Here, $x := e$ is a shortcut for assignment $(x, []) e$. Computation models of our theory and implementation are given.
Example 5.6.1 Essential occurrences of a forall term.
Consider the following examples in pseudo code:

\[
\begin{align*}
\text{forall } i=0,n \ j=0,m \quad &\quad \text{forall } i=0,n \ j=0,m \quad &\quad \text{forall } i=0,n \ j=0,m \\
A[i,j] = expr \quad &\quad A[i,j] = F(X) \quad &\quad A[i+1,j] = F(X) \\
\text{end} \quad &\quad \text{end} \quad &\quad \text{end}
\end{align*}
\]

(1) \quad (2) \quad (3)

where the expressions \( n \) and \( m \) are not dependent on each other, \( expr \) is some arbitrary expression that does not contain a function call and is not dependent on \( A \), \( F \) represents a function, and \( X \) is an actual argument list that is not dependent on the data structure \( A \).

In each of the languages Booster, CM Fortran, and HPF the index space over which is iterated is the Cartesian product \([0 \ldots n] \times [0 \ldots m]\).

CM Fortran In CM Fortran, Example 5.6.1(1) will cause the assignments to be executed on the CM in parallel. The assignments of Example 5.6.1(2) will be executed sequentially because of the function call on the right hand side. Example 5.6.1(3) is not valid since CM Fortran allows only one statement in a forall body.

Booster In Booster, both Example 5.6.1(1) and Example 5.6.1(2) will perform the assignments in arbitrary order. Because in Booster functions are referentially transparent, the function call causes no side effects, and therefore it is guaranteed that each element is used as a source before it is used as a target. In Booster too, only one assignment is allowed in the forall body, which makes Example 5.6.1(3) invalid.

HPF In HPF, Example 5.6.1(1) and 5.6.1(2) have the same meaning as in Booster. Although pure functions in HPF need not be referentially transparent, it is forbidden for those functions to have side effects. This allows the different body instances of a forall statement to be evaluated in arbitrary order. Example 5.6.1(3) is semantically equivalent to the following consecutive forall terms:

\[
\begin{align*}
\text{forall } i=0,n \ j=0,m \quad &\quad \text{forall } i=0,n \ j=0,m \\
A[i,j] = expr, \quad &\quad A[i+1,j] = F(X)
\end{align*}
\]

Note that the second forall term only starts when the first forall term has finished. It cannot be rewritten to one independent do loop, because a define-define dependency exists for \( A[i,j] \), \( 1 \leq i \leq n-1 \) and \( 0 \leq j \leq m \).
**V-nus**  Example 5.6.1(1) interpreted in *Booster*, CM Fortran, or HPF can be represented in *V-nus* by:

```
forall [(i, (n,+1)), (j, (m,+1))] statements [
    assignment (A, [i,j]) expr
]
```

Example 5.6.1(2) interpreted in CM Fortran needs a sequential loop in *V-nus*, such as:

```
iteration [(i, (n,+1)), (j, (m,+1))] statements [
    assignment (A, [i,j]) functioncall F [X]
]
```

In *Booster* and HPF this example can be represented in the same way as Example 5.6.1(1) is represented. Example 5.6.1(3) interpreted in HPF can be rewritten to two single assignment *forall* terms as presented above. These can easily be translated to *V-nus*. Note that if Example 5.6.1(3) was interpreted in *V-nus* directly, it would denote a non-deterministic *Forall* term because of the define-define dependencies. Define-define dependencies are allowed if they occur in the same body-instance. For example, if the subscript i:1 of Example 5.6.1(3) is replaced by i then the *Forall* term has become deterministic.

**HPF directives**  Every independent do loop in HPF can be represented by the *V-nus Forall* term, since no loop carried dependencies occur at all. Due to *V-nus* pragmas the effectuality of the independent directive can also be utilized.

The *new* directive in HPF is used to create variables that are local to a single body-instance. In the semantic model of *V-nus* it is not possible to use loop bodies as scope boundaries. So, the named variables in the *new* directive of HPF cannot be represented in *V-nus* by locally declared variables in a loop. As mentioned earlier in this thesis, the possible kinds of scope boundaries may change in future versions of *V-nus*; then the *new* directive can have its representative in *V-nus*.

**Functions**  Since *V-nus* requires functions to be referential transparent, functions of other languages that are less restrictive need to be rewritten in *V-nus*. If a non-*V-nus* function uses (or writes to) a global variable, it can be represented by a corresponding *V-nus* function where this global variable is passed via another function parameter (and consequently becomes local to the function). As a result, an HPF *forall* term with a call in its body to a pure function that uses a global variable can be represented in *V-nus* while fully preserving the semantics and effectiveness.

**V-nus specific expressiveness**  Now, we show an example of an optimization that can only be expressed by using the *V-nus Forall*. Consider the following matrix operation (for readability the assignment is denoted in pseudo code):
iteration [(i,m)] statements [
    forall [(i,n)] statements [
    ]
]

The optimization we have in mind is based on synchronization elimination [45]. By exchanging the i and j loop the operation can be expressed as

forall [(i,n)] statements [
    iteration [(j,m)] statements [
    ]
]

which gives no computational differences in the result. Instead of executing *Forall* terms in sequence, the *Forall* body instances can now be executed concurrently, yet obeying the j sequence. It is easy to see that no define-define dependency occurs, which makes it a deterministic *Forall* term. This *Forall* term is not well-formed in the other parallel languages mentioned in this section.

## 5.7 The semantics of *Foreach*

As with certain *Forall* terms, there are certain *Foreach* terms that may have a non-deterministic behaviour. We do not go into the conditions that make a difference between a deterministic *Foreach* and a non-deterministic *Foreach*. To compute the meaning of a *Foreach* term, any order can be chosen in which the different body instances are executed. Actually, we should say 'a meaning' of the *Foreach* term, since the outcome of the *Foreach* may depend on the order. In our semantic model we will not compute program states for all possible execution orders. We simply use an arbitrary execution order that defines the outcome of a *Foreach* term. In case it is necessary to know whether the *Foreach* is deterministic or not, one needs to repeatedly compute the semantics of this *Foreach* term for different execution orders and compare the results.

The *Foreach* term can be used as a 'reduce' operation. Consider, for instance, a summation of array values $X_i$. Suppose that these $X_i$ values will be computed on different processes $p_i$, and will then be sent to some 'central' process $p$ that adds all values together. This central process can express the summation as presented in Example 5.7.1.
Example 5.7.1 The Foreach expresses a reduce operation.

\[
\text{foreach } [(i, n)] \text{ statements [ receive } p_t(X, [i]),
\text{ assignment } (\text{SUM, } [ ]) ((\text{SUM, } [ ]), +, (X, [i]))\]
\]

It does not matter in which order the \( X_i \) are summed. In this way, this Foreach term has very much in common with a reduce operation.

The final result of the summation (SUM) does not depend on the order in which the body instances are executed. However, the efficiency heavily depends on the order of body instance execution. Since the Foreach allows any order, it can be left to the run-time system to decide which order is chosen; the most efficient order will be the order of arrival of the \( X_i \) elements on the central process.

Of course, efficiency might improve when partial sums are computed on different processes. Such a strategy is hard to express using the Foreach term. The Foreach term is most suited for situations where partial computations are contracted out to other processes, and where the results finally need to be collected by a single process in any order. The operation that ‘collects’ the results should not be time consuming, otherwise it might be better to distribute this computation as well. In that case it is more efficient to use the Reduce expression (see Section 5.8).

The semantics of the Foreach term are very similar to those of the Iteration term. For the latter, we introduced the \( \leq_{\Phi} \) order that determined the subsequent body instances. The semantic function for a sequential loop (SL of Definition 4.7.6) was then defined by using the minimum of ordered index states with respect to this \( \leq_{\Phi} \) order. For the Foreach there is no need to define some order first. We can directly define the function \( \text{takeOne}_{\Phi} \) (the counterpart of \( \text{min}_{\Phi} \)) that takes one element of a set of ordered index states. The function that is used to describe the semantics of the Foreach is then defined in the same way as the SL function, except that it uses the \( \text{takeOne}_{\Phi} \) function instead of the \( \text{min}_{\Phi} \) function.

(For simplicity we use the \( \leq_{\Phi} \) order in the definition of \( \text{takeOne}_{\Phi} \). This is only needed to express exactly one chain of ordered index states of which one is taken. However, any chain would suffice as long as the chain notation is unambiguous.)

Definition 5.7.1 Let \( T_k = \{ t \mid t \text{ is an ordered representation of an arbitrary } \varphi \in \Phi, |\varphi| = k \} \) for an arbitrary \( k \in \mathbb{N} \) be the set of ordered index states of length \( k \). Let \( T \subseteq T_k \) be an arbitrary subset of \( T_k \). Let \( t_0, \ldots, t_n \) be a chain such that \( t_i \leq_{\Phi} t_{i+1} \) for all \( 0 \leq i < n \) and \( t_i, t_n \in T \). Let \( (j_0, \ldots, j_n) \) be a predefined permutation of \( (0, \ldots, n) \). The function
takeOne : \mathbb{P}(T_k) \rightarrow T_k is defined as

\[
takeOne_\Phi(T) = t_0
\]

Note: the function \(takeOne_\Phi\) returns the first element of a permuted chain. This is an arbitrary choice; it could be any element of the chain.

Since the Foreach term is in fact a variant of the Iteration term, a function \(SL'\) is introduced which is a variant of the function \(SL\).

**Definition 5.7.2** Let \(T_k = \{ t | t \) is an ordered representation of an arbitrary \(\varphi \in \Phi, |\varphi| = k\}\) for an arbitrary \(k \in \mathbb{N}\) be the set of ordered index states of length \(k\). The function \(SL' : \mathcal{L}(Statements) \rightarrow \mathbb{P}(T_k) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \Sigma\) is defined as follows.

\[
SL'(Statements)(T)(\delta)(\varphi)(\sigma) = \begin{cases} 
\sigma & \text{if } T = \emptyset \\
SL'(Statements)(T \setminus \{ t \})(\delta)(\varphi)(\sigma') & \text{otherwise}
\end{cases}
\]

where

\[
\begin{align*}
t &= takeOne_\Phi(T) \\
\varphi' &= \varphi \smallsetminus \langle t \rangle \\
\sigma' &= \mathcal{M}(Statements)(\delta)(\varphi')(\sigma)
\end{align*}
\]

The set \(T\) represents a set of ordered index states. The ordered index state \(t\) is the arbitrarily chosen element of \(T\) as defined by the function \(takeOne_\Phi\). The ordered index state \(t\) can be taken only once, since it is removed from \(T\) in the next recursive call to \(SL'\). Similar to the definition for the \(SL\) function, the index states \(\varphi'\) are not cumulatively changed during the recursive calls to \(SL'\), but the program states \(\sigma'\) are. The meaning of a Foreach term is given by the following extension to the meaning function \(\mathcal{M}\).

\[
\mathcal{M}(Foreach)(\delta)(\varphi)(\sigma) = SL'(Statements)(T)(\delta)(\varphi)(\sigma) \quad \text{where} \quad \begin{align*}
\mathcal{Foreach} &= \quad \text{foreach} \\
Cardinalities &= \quad \text{Cardinalities} \\
\mathcal{D} \mathcal{P} &= \quad \text{Statements} \\
T &= \quad \mathcal{D} \mathcal{P}(\text{Cardinalities})(\delta)(\varphi)(\sigma)
\end{align*}
\]

### 5.8 Other implicit parallel constructs

In many data parallel languages, constructs exist that express so called ‘reduce’ and ‘scan’ operations. A reduce operation applies a given operator to the elements of a data structure, and the result is a scalar. The scan operation, too, applies an operator over the elements of a data structure, but the result is a data structure with the same ranks as the original data structure.

The reduce operation in V-nas can be expressed as: \text{reduce} \otimes V, where \(\otimes\) represents a commutative, associative operator, and \(V\) is a View. If \(V\) consists of the elements
V_0, \ldots, V_{n-1} \text{ then this reduce operation means: } V_0 \bigotimes \ldots \bigotimes V_{n-1}. \text{ As such, the reduce operation belongs to the class of Expression, and the denotational semantics are defined by the following extension to the known function } \mathcal{E}.

\[
\mathcal{E}(\text{Reduce})(\delta)(\varphi)(\sigma) = \mathcal{E}(\text{Structure})(\delta)(\varphi \varphi_0)(\sigma) \bigotimes \ldots \bigotimes \mathcal{E}(\text{Structure})(\delta)(\varphi \varphi_{n-1})(\sigma)
\]

where

\[
\begin{align*}
\text{Reduce} &= \text{reduce } \otimes \text{(Cardinalities, Structure)} \\
\varphi_0, \ldots, \varphi_{n-1} &\in \mathcal{D}P(\text{Cardinalities})(\delta)(\varphi)(\sigma)
\end{align*}
\]

and where \( \otimes \in \text{Binary} \) is a associative, commutative operator.

Associativity and commutativity of the operator make it possible to apply the operator on any pair of operands. That is: the run-time system can choose in which order it ‘reduces’ the expression \( V_0 \bigotimes \ldots \bigotimes V_n \). Take a second look at the Foreach term of Example 5.7.1. Suppose that the \( X_i \) are on one and the same process, and the number of \( X_i \) is huge. Then, the presented Foreach term is not suited to do the summation, since a sequential operation has been denoted. A Reduce term like: \( \text{reduce } + ([i,n], (X, [i])) \) is more efficient. The run-time system can choose to compute partial sums on different processes, and to ‘reduce’ these partial sums (see Figure 5.3).

![Figure 5.3: The Reduce allows for computing the b-sum in parallel with the c-sum. The Foreach would perform these computations sequentially.](image)

The scan operation in V-nus is not expressed by a single V-nus term. It needs to be composed by the programmer. The reason is that there are no additional advantages in having a special V-nus term for a scan operation. The scan operation expresses, in fact, multiple reduce operations. A general scan operation in pseudo code looks like (similar to the reduce operation): \( \text{scan } \otimes V \), where \( V \) represents a data structure. If \( V \) consists of the elements \( V_0, \ldots, V_{n-1} \) then the scan operation returns a data structure \( V' \) where \( V'_j \) is the ‘reduce’ of all \( V_j \) where \( j \leq i \). It depends on the locality of the \( V_j \) elements, the number of elements, and the cost of the operator (and possibly other considerations) what the most efficient implementation is to denote a scan operation. We consider two possibilities. The scan operation can be expressed by using a Foreall term. For simplicity, we consider a one dimensional View \( V \) of rank \( n \), where the result of the scan is represented by a View \( W \).
forall \([(i,n)]\) statements
  assignment \((W, [i]) \reduce \otimes ([j,(i,+1)]), (V, [j])\)
]

This scan operation is advantageous when \(n\) is small, and when the operator \(\otimes\) is 'cheap' for the performance (in terms of a performance cost function). In this way, a distributed computation of the different 'reduces' can improve the performance over a sequential computation. Otherwise, the following sequential computation may be more favorable.

assignment \((W, [0]) \ (V, [0])\),
iteration \([(i,(n,-1)])\] statements
  assignment \((W, [[[i,+1]]) \ ((W, [i]) \otimes \ (V, [[i,+1]]))\)
]

Of course, combinations of the first and the second program can also improve the performance under certain conditions. It is left to the programmer to compose a suited scan operation.

### 5.9 Conclusion

We can now show that we have met the four requirements for the generalized forall statement, as listed in Section 5.1.

**denotational semantics** For non-deterministic forall statements an unambiguous program state change is enforced by the specification of a computation order; i.e. the order in which the program states \(\sigma_i\) of the body instances are merged. The program state change of a deterministic forall statement is not dependent on the computation order.

Suppose that \(\text{diff}_i\) and \(\text{diff}_j\) contain the same variable \(x\). That means that both in \(\sigma_i\) and in \(\sigma_j\) the variable \(x\) has been changed with respect to the original \(\sigma\). That can only be so, if both body instances \(i\) and \(j\) have a 'define' for \(x\). But for deterministic forall statements this can never occur. So, there are no two different \(\text{diff}_i\) and \(\text{diff}_j\) containing the same variable. When all the \(\text{diff}_i\) are put into \(\sigma\) for the construction of \(\sigma'\), \(\sigma'\) will be a valid program state. Note that it does not matter in which order the \(\text{diff}_i\) are inserted into \(\sigma\) – the same \(\sigma'\) is constructed.

**efficiency** Computing the difference between two program states can be inefficient when done naively. Therefore, the \(V\)-nus implementation does not use the same computation as given in the semantics. The approach taken here requires some computation overhead compared to a sequential loop. This overhead is due to the following computations:

- Before the body instances can be executed, each body instance must get its own program state (which is usually a small subset of the global program state).
During execution of a body instance, each write action is performed twice. One to update the local program state and one to update the global program state. In many cases, one of these two write actions can be omitted.

The computation time for the construction of the program state $\sigma'$ is in the order of the number of variables that are used in the forall body. A direct implementation of the theoretical scheme would need linear time in the number of variables of the entire program and the number of body instances of the forall statement.

Pragmas can be used to help the compiler such that no unnecessary copies of variables are made. This reduces computation and space overhead.

**representation** $V$-nus can be used to capture the meaning of different definitions of forall statements, as is shown in Section 5.6. Therefore, we think that our forall definition is suitable for an intermediate representation.

**expressiveness** When a programmer specifies a forall statement, only one extra condition has to be taken care of compared to a sequential loop: the same variable may not be written to by two (or more) body instances. We think that this concept can easily be applied when programming forall statements. However, the programmer must be able to verify whether the condition is met, such that non-determinism can be detected. Partially, this can be done at compile-time. A run-time solution for the other cases requires too much overhead in general. But while using execution trace techniques it is possible to recognize a define-define dependency, when different values are written to the same variable. When the same value is written twice to that variable a define-define dependency is not recognized, but nevertheless the result is deterministic. So, a weaker condition can be checked: the execution trace of a program has no define-define dependency that causes non-determinism.

The For each and Reduce terms fit in where the For all term 'fails'. In case subresults need to be computed cumulatively in the body instances the For all term cannot be used. The For each term can then be used, and is suited to specify reduction computations. The advantage over the Reduce term is that the For each allows more complex computations for each reduction step. The disadvantage, however, is that the compiler will generate rather simple levels of parallelism, if it generates code for parallelism at all. Where the reduction can be expressed by a binary, associative, commutative operator the Reduce term will be most efficient. Now, multiple levels of parallelism – trees of processes – can be introduced by the compiler.

We think that the set of implicit parallel language constructs introduced in this chapter (the For all, the For each, and the Reduce) gives sufficient support for:

- the ease of programming parallelism.
- the different forms of parallelism.
- the generation of efficient code for parallelism.
Chapter 6

Explicit parallelism

The most beautiful and deepest experience a man can have is the sense of the mysterious. It is the underlying principle of religion as well as of all serious endeavour in art and in science.

My Credo
Albert Einstein

6.1 Introduction

The term ‘explicit parallelism’ refers to the V-nus terms: Send and Receive. These terms ‘explicitly’ require parallel execution of a program. The Send term sends a data structure, and from then on, a Receive term can receive this data structure. Although this sending and receiving might be done by one and the same process, the data structure is really ‘transmitted’ from the ‘Send point’ to the ‘Receive point’. So, there is a mechanism that allows the transmission of data structures. We cannot think anymore of a program that might be executed in parallel, but can also be executed sequentially (like in the previous chapter). Here, we ‘explicitly’ require more than a sequential execution.

One can find the terms ‘interleaved processes’, ‘concurrent processes’, and ‘parallel processes’ in literature (for instance: [5], [11], [39]). Interleaved and parallel processes mainly differ in their operational semantics, which is of no importance in this chapter. Concurrent processes have much in common with our implicit parallelism. In the rest of this chapter, interleaving is also covered when we mention parallel processes, parallelism, etc.

Processes that communicate with each other (interchange data structures) need communication channels. For efficiency considerations it is important to know which processes are directly connected with a communication channel. Among others, the efficiency then depends on the number of processes involved in transmitting a data structure from one process to another. These considerations are not part of this thesis. We simply assume that each process (of a parallel program) can send to and receive from all other processes taken part in the computation.
As is mentioned earlier in this thesis, we use a distributed memory model. So, each process consists of a virtual processing unit and (local) memory. For this reason, the *Barrier* (to induce a synchronization on all processes, as explained in Chapter 2) will not be explained, since its impact on parallel programs is limited to a shared memory model. The *Broadcast* communication term expresses transmission of data structure to all other processes. We see this term as a shortcut for a list of *Send* terms, and do not define formal semantics for it. The communication terms *Want*, *Synchronize*, and *Redistribute* are also beyond the scope of this chapter. *Want* and *Synchronize* can each be represented by a pair of *Send* and *Receive* terms (see also Chapter 7 and Appendix D). *Redistribute* changes the distribution function for a data structure, which has no effect on the meaning (as it is given with our semantics). We go into more detail on these three communication terms in Chapter 7.

Here, we focus on the meaning of processes executing in parallel which may communicate. The idea is simple. Since the architecture of communication channels is irrelevant, the meaning of processes executing in parallel can be reflected by a set of process states. Each process state represents the meaning of a single process. Such a process is actually a sequential program, thus the notion of program state from the previous chapters can be re-used. This familiar program state needs to be extended when semantics must be given to communications.

The model to be presented shortly will not cope with the spawning of processes. It is assumed that the program in question is distributed over some set of existing processes. As such, the primary given number of processes on which the program will execute remains fixed; i.e. processes can not be killed, nor can they be created. The notion of a process we use in this chapter conforms with the one introduced in Chapter 2. So we recall that a process is completely abstracted from the underlying hardware model.

The expressions in $\mathcal{L}(Ownership)$ are not described in our semantic model. These expressions are based on the distribution table. We do not go into the semantics of distributing data over processes. We simply assume that the data has been distributed. As such, we also assume that the value of the expressions of $\mathcal{L}(Ownership)$ can be looked up in the distribution table.

The main problem we focus on in this chapter is the semantics of communicating processes. The communication is enabled with the terms *Send* and *Receive*. Since we strongly prefer our model to remain compositional, we want to give semantics to these terms without the need to consider them in *Send* - *Receive* pairs only. A communication requires two (or more) processes in general; so, focusing on a single process and still being able to give semantics to it asks for further investigation. The solution we have chosen (and is presented in this chapter) has much in common with the approach taken in [73]. The main difference between our context of communicating processes and that of [73] is the type of communication; in the latter synchronous communication is considered, where we have asynchronous communication. More semantic models involving asynchronous communication have been described (see, for instance, [1]), but the one chosen in [73] most resembles our approach.

This chapter is organized as follows. In Section 6.2 the type of a process state is defined.
6.2. THE PROCESS STATE

The meaning of a process, reflected in the process state, is defined in Section 6.3. Here, the \textit{V-nus} terms introduced so far plus the \textit{Send} and \textit{Receive} terms obtain their meaning. The focus is still on a single process. In Section 6.4 the interaction between multiple processes is explained.

6.2 The process state

In this section we introduce the notion of a \textit{process state}, which fulfills the same kind of role as the program state does. Whereas the program state (together with the index state and the declaration state) represents the meaning of a program that is executed sequentially, the process state reflects the meaning of a single process in a context of multiple processes executing in parallel. Actually, a process is a program executing sequentially, but is ‘aware’ of other processes; i.e. processes are sequential programs that can communicate with each other. This means, the meaning of a process can be computed in the same way as described in Chapter 4 and Chapter 5, extended with (mathematical) constructs that cope with the meaning of communications.

Before we start defining the mathematical constructs needed to cope with process communications, we shortly give an overview of the concepts to be defined. The process state we are about to construct will have a type called $\Gamma$. The type $\Gamma$ will be a tuple space, where each tuple contains a declaration state ($\Delta$), an index state ($\Phi$), a program state ($\Sigma$), and an additional state, called the \textit{communication state}. The communication state is of type $\Gamma_B$, which will represent a ‘buffer of packets’. A packet is a tuple representing what data element has been transmitted between which two processes. These processes are uniquely identified with a process number. As such, a buffer of packets is a record of data elements that have been sent or received.

We start with the definition of the set of process numbers.

\begin{definition}
Let $P \subseteq \mathbb{N}$ be the set of process numbers. According to some coding formalism, each $pnr \in P$ uniquely identifies a process.
\end{definition}

The set of ‘packets’ can now be defined as follows.

\begin{definition}
\begin{align*}
\text{Packet} & \ = \ P \times P \times \text{Data}
\end{align*}
\end{definition}

A packet $(f, t, d) \in \text{Packet}$ must be interpreted as: the data element $d$ is sent from process $f$ to process $t$. The communication state (or: a buffer of packets) is simply a tuple of elements of type $\text{Packet}$. (Remember the notation $X|_n$ of Definition 4.4.1, denoting tuples over $X$ of length $n$.)
Definition 6.2.3 Let $\Gamma_B$ be the type of the communication state (a buffer of packets), defined as

$$\Gamma_B = \{b \mid b \in \text{Packet}_n, n \in \mathbb{N}\}$$

We are now able to define the set of process states.

Definition 6.2.4 The set $\Gamma$ of process states (with typical elements $\gamma, \gamma', \ldots$) is defined as:

$$\Gamma = \Delta \times \Phi \times \Sigma \times \Gamma_B$$

Extra requirement for $\text{Loc}$: Remember that a specific element $\ell$ was assumed such that $\sigma_L(\ell)$ represents the current scope (for any program state $\sigma$). In case we are dealing with processes, we assume another specific element $\varphi$ in order to return the 'current process number'. So, for the set of location functions, specified in Definition 3.3.2, we add the assumption that a specific element $\varphi \in \text{Loc}$ exists, such that $\sigma_L(\varphi)$ has the current process number as result, for all $\sigma \in \Sigma$. The current process number is the number that identifies the process that uses this program state.

All 'states' we have introduced so far, still do not represent the meaning of a program running in parallel. Parallel execution requires multiple processes, and as a consequence, a 'state' representing such parallelism must consist of multiple process states. Therefore, we introduce a notion of a parallel program state, which is simply a set of process states.

Definition 6.2.5 Let the set of parallel program states $\Psi$ (with typical elements $\psi, \psi', \ldots$) be defined as

$$\Psi = \mathcal{P}(\Gamma)$$

In the forthcoming sections we will use a meaning function on different 'levels'. When the context consists of a single program that is executed sequentially, the familiar meaning function $\mathcal{M}$ is used. If necessary, we use the subscript $\Sigma$, such that

$$\mathcal{M}_\Sigma : \text{Vnus} \to \Delta \to \Phi \to \Sigma \to \Sigma$$

is the same as $\mathcal{M}$ of Definition B.1.

In case the focus is on a process in a context of other processes, the meaning of a single process is represented by a different meaning function $\mathcal{M}_\Gamma$. For reasons to be explained later in this chapter, the function $\mathcal{M}_\Gamma$ has a V-nus program and a single process state as input, and multiple process states as output. Thus

$$\mathcal{M}_\Gamma : \text{Vnus} \to \Gamma \to \Psi$$
6.3 THE MEANING OF A PROCESS

The meaning of a set of processes running in parallel is described by the meaning function \( \mathcal{M}_\Psi \). Given a \( V\text{-}nus \) program and a set of process states, it determines the resulting set of processes after the \( V\text{-}nus \) program has been executed in parallel over the given processes.

\[
\mathcal{M}_\Psi : \text{Vnus} \rightarrow \Psi \rightarrow \Psi
\]

6.3 The meaning of a process

As mentioned earlier in this chapter, the meaning of a process executing a program mainly depends on the meaning of a sequential program. All \( V\text{-}nus \) terms that have been considered so far get their meaning in the same way as presented in the previous chapters. In this section we describe how the 'old meaning function' (\( \mathcal{M}_\Sigma \)) is incorporated into the 'new meaning function' (\( \mathcal{M}_\Gamma \)) with respect to the familiar \( V\text{-}nus \) terms. Additionally, in this section too, we describe the meaning of the explicit parallel \( V\text{-}nus \) terms: Send and Receive.

We will define the meaning function \( \mathcal{M}_\Gamma \) for processes in three steps: the meaning of \( \text{Send} \), the meaning of \( \text{Receive} \), and the meaning of the already familiar \( V\text{-}nus \) terms. We start with the last step. The complete definition can be found in Appendix B.

Consider all \( V\text{-}nus \) terms of the previous chapters of which the semantics are given by the meaning function \( \mathcal{M}_\Sigma \). Assume that \( PS \) is exactly this set of \( V\text{-}nus \) terms – i.e. the set of terms whose meaning can be represented by a pair of program states.

\[
PS = \{ c_0 \in \text{Vnus} \mid \mathcal{M}_\Sigma(c_0) \neq \bot, c_0 \text{ is a singulary term} \}
\]

The meaning of these terms executed by a process is then described by

\[
\mathcal{M}_\Gamma(c)(\gamma) = \{ (\delta, \varphi, \sigma', \gamma_n) \} \quad \text{if } c \in PS
\]

where

\[
\begin{align*}
\gamma &= (\delta, \varphi, \sigma, \gamma_n) \\
\sigma' &= \mathcal{M}_\Sigma(c)(\delta)(\varphi)(\sigma)
\end{align*}
\]

The statement send From To Expression must be sent from process From to process To. Only the process from which the data structure must be sent needs to take care of this 'send statement'. Note that the Send of \( V\text{-}nus \) is a non-blocking Send. That means that the process that sends a data structure \( d \) does not need to wait for the reception of the process to which it is sent. In our semantics this is reflected by putting a 'send packet' in the communication buffer. The meaning function for Send is defined as follows.
\[ M_T(Send)(\gamma) = \begin{cases} \{\gamma\} & \text{if } \sigma_L(\delta) \neq \tilde{f} \\ \{(\delta, \varphi, \sigma, \gamma_B')\} & \text{otherwise} \end{cases} \]

\[
\begin{aligned}
\text{Send} &= \text{send } f \ t \ e \\
\gamma &= (\delta, \varphi, \sigma, \gamma_B) \\
\gamma_B' &= \gamma_B \bullet ((f, \tilde{t}, \tilde{e})) \\
\tilde{f} &= \mathcal{E}(f)(\delta)(\varphi)(\sigma) \\
\tilde{t} &= \mathcal{E}(t)(\delta)(\varphi)(\sigma) \\
\tilde{e} &= \mathcal{E}(e)(\delta)(\varphi)(\sigma)
\end{aligned}
\]

In case the process in question is the process from which the data structure must be sent, the communication buffer is appended with the packet that is transmitted. In the other case, the meaning of a `Send` means ‘nothing’. This is still rather straightforward.

The meaning of a `Receive` is a bit less straightforward. The `Receive` of V-nus is a blocking `Receive`. So, the process can only proceed with its execution when it actually has received what it was waiting for. Two remarks: firstly, in our semantics we do not check whether the data structure that is received complies with what was expected by the `Receive`. We simply assume that if something is sent, it is a valid data structure. Secondly, in our semantics there is no notion of time, nor is there a notion of simultaneous execution. That means that in our semantics a blocking `Receive` does not ‘block’ the computation. The approach taken here is to consider every possible data structure that is a candidate to be sent to the process in question. Later, we reduce this explosion of possible sent packets. This reduction is postponed until the meaning of all processes together is considered; i.e. when the meaning function \( M_\psi \) on multiple processes determines the meaning of a V-nus program.

In a way, this description of a `Receive` does block our semantics. This is due to the fact that ambiguous semantics originate from a `Receive` for a certain process. We come to that again after the following definitions for the meaning of a `Receive`. Note that data structures can only be received into existing program variables. This reception of a data structure \( d \) into an existing program variable \( Id \) is described by the function `Assign`. The function is only needed to facilitate the definition of the meaning of `Receive` more easily.

**Definition 6.3.1** Let \( d \in Data \) be an arbitrary data structure. Let \( \sigma \in \Sigma \) be a program state where \( Id \) is an existing view identifier such that the domain of \( \sigma_V(Id, s) \) is a spanning bounded set for \( d \). Let \( s \) be the current scope \( \sigma_L(\ell) \). The function \( \text{Assign} : (Data \times VI) \rightarrow \Sigma \rightarrow \Sigma \) is defined as

\[
\text{Assign}(d, Id)(\sigma) = \sigma \circ \sigma_L'
\]

where \( \sigma_L' = \{ (\sigma_V(Id, s)(i), d_i) | i \in \text{dom}(\sigma_V(Id, s)) \} \)

With the aid of the function `Assign` the meaning of `Receive` is defined as
\[ \mathcal{M}_T(\text{Receive})(\gamma) = \begin{cases} \{\gamma\} & \text{if } \sigma_t(\varphi) \neq \bar{t} \\ X & \text{otherwise} \end{cases} \]

Receive = receive t \; \text{Id}

\[
\begin{align*}
\gamma &= (\delta, \varphi, \sigma, \gamma_B) \\
X &= \{\gamma' \mid \gamma' = (\delta, \varphi, \sigma', \gamma'_B), \gamma'_B = \gamma_B \bullet ((\bar{f}, \bar{t}, d)), d \in \text{Data}\} \\
\bar{f} &= E(f)(\delta)(\varphi)(\sigma) \\
\bar{t} &= E(t)(\delta)(\varphi)(\sigma) \\
\sigma' &= \text{Assign}(d, \text{Id})(\sigma)
\end{align*}
\]

Considering all possible data structures that might have been sent (or will be sent) by a different process, gives a set \(X\) of process states. In \(X\) are several process states for which the current process number is one and the same. This is what we meant by 'blocking the semantics'. As long as multiple process states identify the same process, it is not defined what data structure has been received. The meaning function \(\mathcal{M}_\psi\) will solve this. So, computing the semantics can proceed while \(\mathcal{M}_\psi\) can still represent that the execution would be blocked.

Note that this 'expansion' of packets due to a Receive can be seen as a lambda abstraction. In that case, the set \(X\) would be replaced by a single, lambda abstracted, process state: \(\lambda d. (\delta, \varphi, \sigma, \gamma'_B)\) where \(\gamma'_B = \gamma_B \bullet ((\bar{f}, \bar{t}, d))\), etc. Having a lambda expression present in the semantics is a clear sign that something is missing; or rather: that the corresponding execution would be blocked. A lambda application is still needed, specifying the actual data that has been received. We prefer the 'set approach' since it conforms more to the notations we used so far.

As a final assertion we mention that the meaning function \(\mathcal{M}_T\), too, is compositional. Let \(c_1\) and \(c_2\) be two arbitrary \(V\)-nus statements (or lists of \(V\)-nus statements). Then

\[ \mathcal{M}_T(c_1, c_2)(\gamma) = \bigcup_{\gamma' \in \mathcal{M}_T(c_1)(\gamma)} \mathcal{M}_T(c_2)(\gamma') \]

Note that the comma in \(\mathcal{M}_T(c_1, c_2)\) is part of the \(V\)-nus language, and does not separate arguments of \(\mathcal{M}_T\).

### 6.4 Semantics of communications

So far, the meaning of a single process is described. Although processes can send and receive data structures to each other, the semantic descriptions depend only on one process at a time. In this section we define the function \(\mathcal{M}_\psi\) on parallel program states, where we deal with the interaction between the processes.

We say that a parallel program has successfully terminated when each sent data structure has been received, and when each expectation to receive a certain data structure has been fulfilled. In other words: for each \textit{Send} a 'matching' \textit{Receive} must exist, and for each
Receive a 'matching' Send must exist. This 'matching' means the following (see Figure 6.1):

- if process \( f \) sends a data structure \( d \) to process \( t \) (i.e. a packet \((f, t, d)\) exists in the communication buffer of process \( f \)) then process \( t \) must receive \( d \) from \( f \) (process \( t \) too must have a packet \((f, t, d)\) in its communication buffer).

- (symmetry of the previous item:) if process \( t \) receives a data structure \( d \) from process \( f \) then process \( f \) must have sent \( d \) to \( t \) (i.e. both processes have the packet \((f, t, d)\) in their communication buffer).

- each subsequent Send (or Receive) of process \( f \) must match (as described in the previous items) a following Receive (or Send) of process \( t \).

This matching is formalized with the relation for the partial match \( \bowtie \). For readability of the definition of \( \bowtie \) the following projection abbreviations are introduced. Given a packet, the functions \( \text{To} \) and \( \text{From} \) return the process number of the process to which something is sent and from which something is sent, respectively.

**Definition 6.4.1** The functions \( \text{To}, \text{From} : \text{Packet} \to P \) are defined as

\[
\text{To}((f, t, d)) = t \\
\text{From}((f, t, d)) = f
\]

First, a partial match is defined on two process states. Thereafter, a complete match is defined on a set of process states. The definition of \( \bowtie \) resembles the descriptions in the items above.

**Definition 6.4.2** Let \( \mathbb{N} \to M \mathbb{N} \) be all monotonic ascending functions on \( \mathbb{N} \). Let \( \gamma, \gamma' \in \Gamma \) where \( \gamma_B, \gamma'_B \in \text{Packet}_{|p} \) for a certain \( n \in \mathbb{N} \) be states for processes identified by \( p \) and \( p' \) respectively. Say, for \( 0 \leq i < n \): \( \text{proj}_i(\gamma_B) = b_i \) and \( \text{proj}_i(\gamma'_B) = b'_i \). The relation \( \bowtie \subseteq \Gamma \times \Gamma \) is defined by

\[
\gamma \bowtie \gamma' \iff \begin{cases}
\exists f : \mathbb{N} \to M \mathbb{N} \\
\forall 0 \leq i < n. \text{To}(b_i) = p' \lor \text{From}(b_i) = p' \Rightarrow b_i = b'_{f(i)} \\
\forall 0 \leq i < n. \text{To}(b'_{f(i)}) = p \lor \text{From}(b'_{f(i)}) = p \Rightarrow b_i = b'_{f(i)}
\end{cases}
\]

A set of matching process states means that each pair of states must match. Note that if two processes have sent nothing to each other, and have not expected to receive something from each other, they match. With the match relation, the matching process states can be filtered from a set of all kinds of process states. This filtering is called a complete match over a set of process states. In terms of the previous section: \( \psi \) might be
'blocked' since it can contain multiple process states representing the same process. The complete match removes this 'blocking' such that a non-ambiguous $\psi$ remains.

In the case that we would use lambda abstraction for the semantics of $Receive$ (as mentioned in the previous section), we should not use the term 'filtering', but use 'application' instead. Each lambda abstraction $\lambda d. \gamma$ should be replaced by the application $(\lambda d. \gamma) d'$. Here, $d'$ can be taken by finding the packet $(f, t, d')$ in a non lambda abstracted process state $\gamma'$, given that $(f, t, d)$ is present in the communication buffer of $\gamma$. It is easy to verify that a clear correspondence exists between the 'lambda abstraction approach' and the 'set approach', since each lambda abstraction $\lambda d. \gamma$ represents the set of process states $\{\gamma \mid d \in Data\}$, where $d$ is a data structure as is specified in the involved definitions.

**Definition 6.4.3** A complete match over a set $\psi \in \Psi$ of process states is denoted as $[\psi]_m$, and is defined as follows.

$$[\psi]_m = \{\gamma, \gamma' \in \psi \mid \gamma \equiv \gamma'\}$$

Finally, the semantics of the $Receive$ can be refined. That is, by requiring a complete match over all process states that are part of a parallel program state, packets are only received when these were really sent (and not just assumed to be sent). The meaning function $M_\psi$ on parallel program states is defined for an arbitrary $V$-nus program $V$ as follows

$$M_\psi(V)(\psi) = \left[\bigcup_{\gamma \in \psi} M_\Gamma(V)(\gamma)\right]_m$$
It can be shown that the outcome of $\mathcal{M}_V(\psi)$ is ‘valid’; that is, only one process state per process number is present in the outcome. Due to the semantics of a Receive more process states come into existence than initially were given. These added process states represent processes that already are represented by a process state. But in the end, the complete match will return a valid set of process states (is: a valid parallel program state).

**Theorem 6.1** For any V-nus program $V$, and a valid parallel program state $\psi \in \Psi$:

$$\forall \gamma = (\delta, \varphi, \sigma, \gamma_B), \gamma' = (\delta', \varphi', \sigma', \gamma'_B) \in \left[ \bigcup_{\gamma'' \in \psi} \mathcal{M}_V(\psi)(\gamma'') \right] : \sigma_L(\psi) = \sigma'_L(\psi) \Rightarrow \gamma = \gamma'$$

\(\Box\)

For readability of the proof, we introduce the following notations, which are only used in the proof below.

- For a term $c \in \textbf{Vnus}$ the expression $c \in V$ is true, if $c$ is present in program $V$. Otherwise, the expression is false.

- For a tuple $t$, the expression $x \in t$ is true, if a projection $\text{proj}_i$ exists ($i \in \mathbb{N}$) such that $x = \text{proj}_i(t)$.

**Proof** Assume $\sigma_L(\psi) = \sigma'_L(\psi)$. We have that $\delta$ and $\varphi$ are fixed under the operation of a complete match; $\sigma$ and $\sigma'$ are equal since $\psi$ is valid.

**Case 1:** No $r = \text{Receive}$ exists such that $r \in V$. Then by Definition B.2 of $\mathcal{M}_V$ the communication buffers are empty: $\gamma_B = \gamma'_B = ()$. Thus: $\gamma = \gamma'$.

**Case 2:** An $r = \text{Receive}$ exists such that $r \in V$. Suppose $\gamma \neq \gamma'$. That means $\gamma_B \neq \gamma'_B$. Say $(f, t, d) \in \gamma_B$ and $(f, t, d) \notin \gamma'_B$. Thus $\sigma_L(\psi) = t$ and $\sigma'_L(\psi) \neq t$. We obtain: $\sigma_L(\psi) \neq \sigma'_L(\psi)$. Contradiction. So $\gamma = \gamma'$.

\(\Box\)
Chapter 7

A calculus of transformations

[Freud:] All this may give you the impression that our theories amount to a species of mythology and a gloomy one at that! But does not every natural science lead ultimately to this — a sort of mythology?

Why War?
Einstein and Freud

7.1 Introduction

We recall the separate phases of the compiler framework we are using. In the previous chapters we have shown the different levels of abstraction that can be used in V-nus. As such it is possible to aim at a front-end that translates high-level data parallel languages to V-nus, leaving the declarative data parallel specifications in tact. The back-end may translate V-nus programs where, for instance, parallelism is denoted with explicit parallel language constructs. As introduced in Chapter 1 a set of engines will be responsible to do the transformations that are necessary to change the abstraction level of a V-nus program. These engines and the information sources they need are called together: a calculus. This chapter introduces a calculus called V-cal (View-calculus).

A more detailed introduction to V-cal is presented in Section 7.2. Guided by an example, Section 7.3 shows some results of applying V-cal rules to an Iteration. The rules themselves are defined in Section 7.4. It is in this section that we present rules that introduce explicit parallel language constructs. It is then easy to see that these rules need guidance in order to cause optimizations. Section 7.5 introduces a 'strategy' that will prevent that the rules thwart each other. In the end, in Section 7.6 we consider an existing library of rewrite rules. It is shown how the language V-nus can easily make use of such libraries. We conclude this chapter with a small discussion on the presented calculus in Section 7.7.
7.1.1 Notational conventions

- We will often make use of an arbitrary concatenation of $V$-nus terms. Since we have no non-terminal in the $V$-nus grammar that defines this concatenation, we use the notation: $\text{Statement}^*$ as a shortcut for the EBNF grammar syntax: $\text{Statement} // \cdot$ (An explanation on this grammar syntax can be found in Appendix A.)

- Consider a concatenation of statements $stms \in \mathcal{L}(\text{Statement}^*)$. If a specific statement $s$ is present as a part of $stms$, we write $stms \setminus s$ for the concatenation of statements where $s$ is removed.

- When introducing extra $V$-nus terms into an existing program (fragment) a notion of 'free labels' is needed. The set $\text{FLAB}$ denotes the set of free labels; i.e. labels that are not used in the currently considered program (fragment).

7.2 The calculus $V$-cal

The view calculus $V$-cal came into existence due to the creation of the language Booster. $V$-cal is intended to be a calculus for supporting the compilation of a high level data parallel language like Booster. The rules in the previous version of $V$-cal gave a meaning to Booster terms; the current version of $V$-cal (used in the rest of this chapter) describes transformations on $V$-nus terms, but the meaning of these terms is prescribed by the denotational semantics. The language $V$-nus is used to express the program patterns on which the calculus will work. In the calculus we have a notion of a rewrite rule, also called transformation rule, where the left hand side specifies a program pattern while the right hand side defines its replacement. We say that a rule matches a program construct if the rule is defined for this program construct. If a rule matches a program construct it will replace this program construct by that of the right hand side. Of course the program, after rewriting, will be semantically equivalent with the original program.

$V$-cal can be divided into three classes of rewrite rules. The first class of rewrite rules is the set of rules that will initially be used when $V$-cal is applied to a $V$-nus program. This set of parallelism identifying rules will be used to rewrite program constructs such that parallelism can be improved. Construct substitutions and loop distributions are examples of this class of rules. The second class of rules in $V$-cal will handle the use of communication statements. These communication handling rules will introduce, move, or remove statements needed for the transport of data. In this class data distribution information will not be used and therefore the communication statements only specify at what point in the program data is needed, synchronization is required, or a redistribution has to be performed. Finally, in the topology dependent rules information can be used about the ownership of data, the data distribution, and the topology.

A calculus specification consists of a language definition (i.e. $V$-nus) and the use and definition of rewrite rules (i.e. the $V$-cal rules). We aim at achieving an engine that reads a program and a specification of a calculus, and will result in a modified, but semantically
unchanged program. This is depicted in Figure 1.2 of Section 1.5. Note that the definition of the language is an integral unit of the specified calculus.

7.3 Effects of using V-cal

Based on the language V-nus a set of transformation rules can be used in order to replace certain program constructs by semantically equivalent program constructs. The calculus V-cal consists of a set of transformation rules and a strategy that prescribes the use of the rules. We will illustrate the use of some V-cal rules of the first class, the parallelism identifying rules, by a small demonstration, based on an example from [102], where sequential code is transformed to code suited for the purpose of parallelism. Below, the intermediate results are presented after each application of a transformation rule. We start with the following program; it is initially presented in pseudo code to help the reader parse the subsequent V-nus syntax.

\[
\text{FOR } i=0 \text{ TO } 99 \text{ DO} \\
\quad x := 5+i; \\
\quad A \ [x] := B \ [x+1] + C \ [x]; \\
\quad E \ [i] := F \ [i+1] \ast A \ [x]; \\
\text{END;}
\]

In translating this to V-nus we now add the optional Labels (here: s1, s11, s12, \ldots) to the Actions, which we have not used so far. These labels become important during data dependency analysis, as will become clear in the succeeding sections.

We obtain the following Iteration:

\[
\text{iteration } [(i,100)] \text{ statements [} \\
\quad \text{assignment } (x, [\ ]) \ (5, +, i) \ s11, \\
\quad \text{assignment } (A, [x]) \ ((B, [(x, +, 1)]), +, (C, [x])) \ s12, \\
\quad \text{assignment } (E, [i]) \ ((F, [(i, +, 1)]), \ast, (A, [x])) \ s13
\]

Applying scalar forward substitution to this V-nus program we get:

\[
\text{iteration } [(i,100)] \text{ statements [} \\
\quad \text{assignment } (x, [\ ]) \ (5, +, i) \ s11, \\
\quad \text{assignment } (A, [(5, +, i)]) \ ((B, [(5, +, i), (i, +, 1)]), +, (C, [(5, +, i)])) \ s12, \\
\quad \text{assignment } (E, [i]) \ ((F, [(i, +, 1)]), \ast, (A, [(5, +, i)])) \ s13
\]

If we consider this as the whole program (so, no references to x exist) we may apply dead code elimination. The result is:
iteration [(i,100)] statements [  
  assignment (A, [(5,+i)]) ((B, [(5,+i),+1]), +, (C, [(5,+i)]))) s12,  
  assignment (E, [i]) ((F, [(i,+1)]), *, (A, [(5,+i)]))) s13  
] s1

The expression evaluator can reduce some expressions at compile time such that the program may be replaced by:

iteration [(i,100)] statements [  
  assignment (A, [(5,+i)]) ((B, [(6,+i)]), +, (C, [(5,+i)]))) s12,  
  assignment (E, [i]) ((F, [(i,+1)]), *, (A, [(5,+i)]))) s13  
] s1

By using data dependency information we can perform a loop distribution with the following result:

iteration [(i,100)] statements [  
  assignment (A, [(5,+i)]) ((B, [(6,+i)]), +, (C, [(5,+i)]))) s12  
] s1,
iteration [(i,100)] statements [  
  assignment (E, [i]) ((F, [(i,+1)]), *, (A, [(5,+i)]))) s13  
] s2

Again by using data dependency information we can replace both loops by a parallel loop such that we end up with:

forall [(i,100)] statements [  
  assignment (A, [(5,+i)]) ((B, [(6,+i)]), +, (C, [(5,+i)]))) s12  
] s1,  
forall [(i,100)] statements [  
  assignment (E, [i]) ((F, [(i,+1)]), *, (A, [(5,+i)]))) s13  
] s2

Of course, it can be questioned whether the original Iteration should be split before the calculus tries to transform the Iterations to Foralls. There is no define-define dependency between different body instances, so one has a (denotational) semantically equivalent program in case the program finally ended up in only a single Forall (see Chapter 5). This is, however, of no consequence for showing the mechanism of using V-cal rules. Since this example is borrowed from Zima [102], we stick to these rewrite steps.

7.4 The specification of transformation rules

For the implementation of V-cal we make use of the data dependency information. Techniques to compute this kind of information are described in, for instance, [65, 102] and are out of the scope of this thesis.
7.4. THE SPECIFICATION OF TRANSFORMATION RULES

The transformation rules can be divided into three classes as is explained in Section 7.2. Examples of the first class have been presented in Section 7.3. We will start showing how we can incorporate transformation rules like ‘loop distribution’ and ‘construct substitution’ in \( V-cal \). Consider the loop distribution function \( LD : \mathcal{L}(\text{Statement}) \rightarrow \mathcal{L}(\text{Statement}^*) \) defined as:

\[
LD(\text{iteration Cardinalities Statements } s) \triangleq \\
\text{iteration Cardinalities statements } [ \text{block1 } ] s_1 , \\
\text{iteration Cardinalities statements } [ \text{block2 } ] s_2 \\
\text{if } \text{Distributive(Cardinalities, block1, block2)}(\text{ddi}) \\
\text{where } s_1, s_2 \in \mathcal{F LAB} \text{ and Statements } = \text{statements } [ \text{block1 }, \text{block2 } ]
\]

The function \( \text{Distributive} \) determines whether the given loop distribution is semantically valid or not. For such a computation the data dependency information is needed which is represented by the variable \( \text{ddi} \) (see also [102]). So \( \text{ddi} \) is input for \( \text{Distributive} \); we abstract from the type of \( \text{ddi} \).

In the same way we can define a transformation rule that replaces an \text{Iteration} with a \text{Forall}. The construct substitution function \( CS : \mathcal{L}(\text{Statement}) \rightarrow \mathcal{L}(\text{Statement}) \) will perform such a replacement in the following way:

\[
CS(\text{iteration Cardinalities Statements } s) \triangleq \\
\text{forall Cardinalities Statements } s \\
\text{if not } \text{DD(Cardinalities, Statements)}(\text{ddi}) \\
\text{and not } \text{DU(Cardinalities, Statements)}(\text{ddi})
\]

Here the functions \( \text{DD} \) and \( \text{DU} \) determine whether a define-define and define-use dependency exist, respectively, between instances of \( \text{Statements} \) for different values of \( \text{Cardinalities} \). As with the loop distribution rule, this dependency can be computed by using the scalar analysis information.

As said before, the second class of transformation rules defines where communication is needed in order to gain parallelism. This kind of parallelism takes care of a correct use of the data needed for a certain computation. Using this class of rules we abstract from the information about the distribution of data. Also the rule that determines the tasks for the individual processes, called a \text{computes-rule}, is not available for these communication handling rules. In the third class of \( V-cal \) rules the data distribution is known and information can be obtained about the owner of certain data. Based on the ownership of data the computes-rule can be specified. For instance, we may specify the ‘owner computes rule’. This means that if the data on the left hand side of an assignment is owned by process \( p \) then the computation of the right hand side will be performed by process \( p \). We use the following communication primitives for the communication handling rules:

- **want \( L \ D \).** This statement denotes that the data structure \( D \) is needed for the execution of the statement indicated by label \( L \). In the third class of \( V-cal \) this statement is translated to \text{Send} and \text{Receive} pairs dependent on the ownership of the wanted data.
• synchronize \(L \ D\). This one is needed to specify that the data structure \(D\) is changed by statement \(L\). A Synchronize is also translated to Send and Receive pairs in the third class of \(V\)-cal when the computes-rule is defined.

• redistribute \(I d\). When redistribution of data is needed, this statement can be used to specify that the data structure \(I d\) needs to be redistributed.

The communication primitives are introduced when processing the \(V\)-nas terms Assignment and ViewStatement. The \(V\)-cal rule that inserts the mentioned communication primitives is the communication insertion function \(CI : \mathcal{L}(\text{Statement}) \rightarrow \mathcal{L}(\text{Statement}^*)\), defined as follows:

\[
CI(\text{view } I d \ \text{View } s) \triangleq \\
\text{view } I d \ \text{View } s, \ \text{redistribute } I d \ s_1
\]

\[
CI(\text{assignment } l h s \ \text{Expression } s) \triangleq \\
\text{want } s \ D_1 \ s_1, \ ... , \ \text{want } s \ D_n \ s_n, \\
\text{assignment } l h s \ \text{Expression } s, \\
\text{synchronize } s \ l h s \ s_m
\]

where the data structures \(D_1, ..., D_n\) are used in Expression, and \(s_1, ..., s_n, s_m \in FLAB\).

Here, \(l h s\) represents both the non-terminals Selection and FunctionCall. The labels \(s_i\) are created in order to (be able to) refer to the accompanying terms.

The next set of rules we need consists of moving the communication primitives. An improvement of efficient parallelism can be achieved by reducing the number of communications without reducing the grain of parallelism. For instance, we can lift the Want terms out of a Forall term and let them execute before entering the loop. The Synchronize terms can be executed when the loop has finished. The \(V\)-cal rule for a communication lift \(CL : \mathcal{L}(\text{Statement}) \rightarrow \mathcal{L}(\text{Statement}^*)\) will do the job.

\[
CL(\text{forall } \text{Cardinalities } \text{Statements } s_1) \triangleq \\
\text{want } s \ d \ s_0, \ \text{forall } \text{Cardinalities } \text{Statements}' \ s_1 \\
\quad \quad \text{if } \text{AppearsIn}(s_0, \text{Statements}) \\
\text{where } \text{Statements}' = \text{Statements} \ \backslash \ \text{want } s \ d \ s_0.
\]

The variable \(\text{Statements}'\) represents the statement list \(\text{Statements}\) without the statement \(\text{want } s \ d \ s_0\). The function \(\text{AppearsIn}\) checks whether the term indicated by \(s_0\) appears in \(\text{Statements}\).

The Synchronize terms can be moved backwards in a similar way such that \(CL\) is also defined by

\[
CL(\text{forall } \text{Cardinalities } \text{Statements } s_1) \triangleq \\
\text{forall } \text{Cardinalities } \text{Statements}' \ s_1, \ \text{synchronize } s \ d \ s_2 \\
\quad \quad \text{if } \text{AppearsIn}(s_2, \text{Statements}) \\
\text{where } \text{Statements}' = \text{Statements} \ \backslash \ \text{synchronize } s \ d \ s_2.
\]
When a Synchronize or a Want is found in a Forall it can unconditionally be moved outside the Forall. The semantics of an Iteration require an extra check on dependencies between the statements in this Iteration, before a communication term can be put before or after this Iteration. Suppose a communication term c within an Iteration is necessary for the transport of some data structure d. Say d is changed by a statement s. Then c may be placed outside the loop if d will not be used by another statement (instance) in the loop. The rule performing such a replacement of the communication terms must therefore check for a define-use dependency as follows:

\[
\text{CL(iteration Cardinalities Statements s₁)} \triangleq \\
\quad \text{want } s \ d \ s₀, \text{ iteration Cardinalities Statements'} s₁ \\
\quad \text{if } \text{AppearsIn}(s₀, \text{Statements}) \text{ and } \\
\quad \text{there does not exist an } s' \in \text{Statements}. \text{DU(Cardinalities, s, s')(ddi)}
\]

where s as well as s’ are labels and
\[
\text{Statements'} = \text{Statements} \setminus \text{want } s \ d \ s₀.
\]

The function DU derives whether a define-use dependency exists between term s and term s’ for all values of Cardinalities.

In the same way we can define the transformation for placing a Synchronize after an Iteration.

\[
\text{CL(iteration Cardinalities Statements s₁)} \triangleq \\
\quad \text{iteration Cardinalities Statements'} s₁, \text{ synchronize } s \ d \ s₂ \\
\quad \text{if } \text{AppearsIn}(s₂, \text{Statements}) \text{ and } \\
\quad \text{there does not exist an } s' \in \text{Statements}. \text{DU(Cardinalities, s, s')(ddi)}
\]

where s as well as s’ are labels and
\[
\text{Statements'} = \text{Statements} \setminus \text{synchronize } s \ d \ s₂.
\]

Note that the define-use dependency check may be too restrictive for some loops. In case the ‘define’ and ‘use’ are carried out by the same process, it is possible to place the communication statements outside the loop. However, the function CL then has to check for the owner of the specific data. This kind of program transformations will not be performed in this class of the calculus.

We are aiming for large program segments that can execute in parallel without any interaction. In order to achieve this we want to use rules that separate a Want - Synchronize pair as far as possible. So, next we present how Want and Synchronize terms may skip over other terms. For now, we only focus on an upward move of the Want in the V-nus program, and a downward move of the Synchronize. We define a communication move rule \(CM: \mathcal{L}(\text{Statement}*) \rightarrow \mathcal{L}(\text{Statement}*)\) as follows:

\[
CM(\text{Action } s₁, \text{ want } s \ d \ s₂) \triangleq \\
\quad \text{want } s \ d \ s₂, \text{ Action } s₁ \\
\quad \text{if not } DU([], s₁; s₂)(ddi)
\]
Remark: *Cardinalities* play no role in this rule; the omission of *Cardinalities* is treated as an empty list of cardinalities in the function *DU*.

The opposite move for a *Synchronize* is then defined as:

\[
CM(\text{\textit{synchronize} } s \ d \ s_1 , \ \text{Action} \ s_2) \triangleq \\
\text{Action} \ s_2 , \ \text{\textit{synchronize} } s \ d \ s_1 \\
\text{if not } DU([], \ s_1, s_2)(ddi) \text{ and not } DD([], \ s_1, s_2)(ddi)
\]

Since a *Synchronize* may cause storing a value of some data structure in a certain memory location, we see the data structure, used by a *Synchronize*, as a ‘define’ of that data structure. Of course it is necessary to check for a define-use dependency and a define-dependency in the above *V-cal* rule.

### 7.5 Applying transformation rules

In order to be able to apply rules to a *V-nus* program we need some mechanism to express how these rules will be applied. Therefore we introduce a notion of a *strategy*. The description of a strategy prescribes which transformation rules will be used and how these are applied (order, kind of tree-walk, etc.).

In general we do not want to perform just any of the possible *V-cal* rules. For instance let us look closer at the communication rule *CM* for the statements *want* *s* *d* *s*\(_1\) and *want* *s’* *d’* *s*\(_2\) where there is no define-use dependency between these statements.

\[
CM(\text{\textit{want}} \ s \ d \ s_1, \ \text{\textit{want}} \ s’ \ d’ \ s_2) = \ \text{\textit{want}} \ s’ \ d’ \ s_2, \ \text{\textit{want}} \ s \ d \ s_1 \\
CM(\text{\textit{want}} \ s’ \ d’ \ s_2, \ \text{\textit{want}} \ s \ d \ s_1) = \ \text{\textit{want}} \ s \ d \ s_1, \ \text{\textit{want}} \ s’ \ d’ \ s_2
\]

After the double application of *CM* we are back where we started. For this reason we must have a strategy for the application of *V-cal* rules that will guide the engine through the permissible rules. Most of the work for this strategy is still an area of active research and is not part of this thesis. We refer to [17] for more information on rule-based compilation.

In our opinion a strategy for *V-cal* encompasses the following elements:

- tree traversals
- rule selections
- matching criteria

The engine will walk several times through the tree as is described in the tree traversals (like the traversals used in attribute grammars [37]): e.g. in depth first order, in breadth first order, in a (static) sweep, or in a (dynamic) visit. Each walk/pass through the tree can be done by using a different traversal. Depending on the pass only a certain selection of the rules will be candidates for application to the tree nodes.

Our engine tries to compute a closure of those rules when applied to the nodes of the tree. The actual application of the rule can be restricted by matching criteria. In addition
7.5. APPLYING TRANSFORMATION RULES

To matching the pattern of the rule, we can also limit the number of times a rule is applied
to a node, and we can also limit the number of nodes to which a rule may be applied.

Now that we have described a strategy, we can apply the rules to a V-nus program.
The tree-walker will try to match a given rule one or several times to parts of the program.
Based on the type of a part of the V-nus program and the signature of the V-cal rules it
can be decided whether applying a rule makes sense or not. The example of using V-cal,
which was given in Section 7.3, resulted in:

\[
\text{forall } [[(i,100)]] \text{ statements } [
\quad \text{assignment } (A, [[(5,+,i)]])([B, [(6,+,i)]], +, (C, [[(5,+,i)]])) s12
\]

] s1.

\[
\text{forall } [[(i,100)]] \text{ statements } [
\quad \text{assignment } (E, [i])([F, [(i,+,1)]], *(A, [[(5,+,i)]])) s13
\]

] s2

This example can now be extended by using a strategy that prescribes to use a ‘communication
insertion’ (CI) for each statement in the program, then tries to use a ‘communication
lift’ (CL) as much as possible, and ends with using a ‘communication move’ (CM) several
times. An intermediate result is:

want s12 ([[i,100]], (B, [(6,+,i)])) t1,
want s12 ([[i,100]], (C, [(5,+,i)])) t2,
forall [[(i,100)]] statements [
\quad \text{assignment } (A, [[(5,+,i)]])([B, [(6,+,i)]],+(C, [[(5,+,i)]]) s12
\]

] s1

synchronize s12 ([[i,100]], (A, [[(5,+,i)]])) t3,
want s13 ([[i,100]], (F, [(i,+,1)])) t4,
want s13 ([[i,100]], (A, [[(5,+,i)]])) t5,
forall [[(i,100)]] statements [
\quad \text{assignment } (E, [i])([F, [(i,+,1)]],*(A, [[(5,+,i)]])) s13,
\]

] s2

synchronize s13 ([[i,100]], (E, [i])) t6

To increase the grain of parallelism statement t4 may be skipped over two statements as
is defined by the ‘communication move’ rule (CM). The above program will then be
transformed into the final form:

want s12 ([[i,100]], (B, [(6,+,i)])) t1,
want s12 ([[i,100]], (C, [(5,+,i)])) t2,
want s13 ([[i,100]], (F, [(i,+,1)])) t4,
forall [[(i,100)]] statements [
\quad \text{assignment } (A, [[(5,+,i)]])([B, [(6,+,i)]],+(C, [[(5,+,i)]]) s12
\]

] s1

synchronize s12 ([[i,100]], (A, [[(5,+,i)]])) t3,
want s13 ([[i,100]], (A, [(5,+,i)])) t5,
for all $[(i,100)]$ statements [  
  assignment (E, [i]) (((F, [i,+,1]),)*,(A, [(5,+,i)])) s13,  
] s2  
synchronize s13 ([(i,100)], (E, [i])) t6

In this example only Forall loops are involved. For the sake of demonstrating the application of transformation rules we chose Forall instances with a single assignment. One can easily verify that no define-define dependency exists between these Forall loops. As such, the original loop (with two assignments) could stay in tact, and with fewer transformation rules one could end up with:

want s12 ([(i,100)], (B, [(6,+,i)])) t1,  
want s12 ([(i,100)], (C, [(5,+,i)])) t2,  
want s13 ([(i,100)], (F, [i,+,1])) t4,  
for all $[(i,100)]$ statements [  
  assignment (A, [(5,+,i)]) ((B, [(6,+,i)]),+.,(C, [(5,+,i)])) s12  
  assignment (E, [i]) ((F, [i,+,1]),)*,(A, [(5,+,i)])) s13,  
] s1  
synchronize s12 ([(i,100)], (A, [(5,+,i)])) t3,  
synchronize s13 ([(i,100)], (E, [i])) t6

A more complex demonstration of V-cal rules handles an Iteration loop. In the next example we show a replacement of some communication statements out of an Iteration. Consider the following V-nus program:

iteration [(i,100)] statements [  
  assignment (A, [i]) (B, [i]) s11,  
  assignment (C, [(i,+,1)]) ((C, [i]), +, (C, [(i,+,2)])) s12  
] s1

Inserting the communication statements into this program by using the V-cal rule CI will result in:

iteration [(i,100)] statements [  
  want s11 ([(i,100)], (B, [i])) t1,  
  assignment (A, [i]) (B, [i]) s11,  
  synchronize s11 ([(i,100)], (A, [i])) t2,  
  want s12 ([(i,100)], (C, [i])) t3,  
  assignment (C, [(i,+,1)]) ((C, [i]), +, (C, [(i,+,2)])) s12,  
  synchronize s12 ([(i,100)], (C, [i])) t4  
] s1

One can easily verify that no define-use dependency exists between s11 and any other statement or statement instance. There is, however, a define-use dependency between the statement instances of s12. So, applying the CL rule several times to this program we see
that the communication statements for s11 can be placed outside the loop. Those for s12 cannot due to the dependencies. But within the loop we can move these communication statements with the CM rule. However, in this case it is questionable whether this will improve the performance. Using the V-cal rules will lead to:

\[
\begin{align*}
\text{want } & s11 \ ((i,100)), \ (B, \ [i])) \ t1, \\
\text{iteration } & ((i,100)) \ \text{statements} \\
& \hspace{1em} \text{want } s12 \ ((i,100)), \ (C, \ [i]) \ t3, \\
& \hspace{1em} \text{assignment } \ (A, \ [i]) \ (B, \ [i]) \ s11, \\
& \hspace{1em} \text{assignment } \ (C, \ [(i,+1)]) \ ((C, \ [i]), \ +, \ (C, \ [(i,+2)])) \ s12, \\
& \hspace{1em} \text{synchronize } s12 \ ((i,100)), \ (C, \ [i]) \ t4, \\
\end{align*}
\]

Note that in case the ‘owner computes rule’ is used, Synchronize statements become ‘dummy’ statements; i.e. they do nothing. That is because the process that changed a data structure \(D\) is also the owner of \(D\), and no additional update for \(D\) (performed by a Synchronize) is needed. Nevertheless, Synchronize statements remain important whether the ‘owner computes rule’ is used or not. One of their uses is to prevent certain statements (for instance, a Want) from moving ‘over’ a Synchronize. Only when this second class of rewrite rules has stopped being applied to a V-nus program, one can eliminate Synchronize statements in case the ‘owner computes rule’ is used.

### 7.6 A case study

V-nus is an intermediate language in a compiler framework in which different tools communicate in this language. The so constructed calculus V-cal is able to incorporate existing tools by translating it to V-nus expressions. An example of such a tool is the Paramat library [63]. In this library, around 150 typical application codes (called patterns) have been collected and also typical modifications (called templates) of them. This combination of a pattern with its template is called a transformation. Such a library is used to look up a certain pattern and replace it by its template. The template can be the optimal code for the pattern on a specific machine.

Since these patterns were selected by examining the source codes of several standard numerical benchmarks programs, we think it is a useful contribution to V-cal. Therefore we will give a demonstration of how this library can be used in our framework. For that purpose we first convert a set of patterns with their templates to V-cal rules. Then we present a simple matrix multiplication which we reduce to a single statement by using the new rules.

In order to do so, we use the language construct LibraryEntry that is used as a statement. These library entries will represent the previously mentioned patterns of the Paramat library. When translating a V-nus program, the library entries will be looked up in some library, and will be replaced by their templates. For example, in [63] an assignment of the
form \( x = c \) may be replaced by a pattern instance \( \text{SInit}(x, c) \) if \( c \) is a constant. Such a replacement can be transformed to a \( V\text{-cal} \) rule \( \text{Sinit} : \text{Statement} \rightarrow \text{Statement} \) as follows:

\[
\text{Sinit}(\text{assignment } x \ c) \triangleq \\
\text{library } [\text{sinit}, x, c] \\
\text{if } \text{Constant}(c)
\]

The function \( \text{Constant} \) tests whether its argument is a constant or not. The original assignment is then replaced by the library entry \( \text{library } [\text{sinit}, x, c] \). When translating the above \( V\text{-nus} \) code, the translation for the library entry is given in a library. In this way also more complex \( V\text{-nus} \) program fragments can be replaced by one constructor. In our back-end we can map these constructors to a semantically equivalent implementation for a given topology.

The hierarchy of the transformation rules in a given library can be used in the ‘strategy’. This means that we can specify a set of rules which the engine will try to match first, after which a second set of rules must be applied. In this way, the pattern hierarchy graph of [63] can be used in order to handle the rules efficiently. We can demonstrate this by a simple vector initialization, based on an example in [63]. Such a vector initialization will recognize consecutive scalar initializations for the vector elements. So, the \( V\text{-cal} \) rule \( \text{Vinit} : \text{Statement} \rightarrow \text{Statement} \) is defined as:

\[
\text{Vinit}(\text{iteration } [(i, n)] \text{ statements } [\text{library } [\text{sinit}, (x, [i]), c]]) \triangleq \\
\text{library } [\text{vinit}, i, n, x, c]
\]

We assume that the ‘\text{Sinit}-rule’ is applied before trying to match the ‘\text{Vinit}-rule’. Having such a requirement coded into the strategy, it makes sense to use \text{sinit} as a subpattern in the \text{Vinit} rule.

Now we will show, by a more complex example, that it is straightforward to use the transformations of the \text{Paramat} library in \( V\text{-cal} \). For this purpose we present a set of \( V\text{-cal} \) transformations that introduces library entries in a \( V\text{-nus} \) program (see Figure 7.1).

The function \( \text{AddMul} \) recognizes the expression where a multiplication is added to some other expression. The \( \text{SSPO} \) rule matches a repeated occurrence of the C idiom \( z \ += \ u \ast v \). In turn, the \( \text{VMO} \) rule looks if a loop body consists of the \text{sspo} pattern, which is a start for a definition of a vector - matrix multiplication. The \( \text{VM} \) rule recognizes a leading initialization of the \( l \)-value vector. Then the \( \text{MM} \) rule checks for a loop around a \( \text{vm} \) pattern and replaces it by a library entry for a matrix - matrix multiplication.

We make use of the function \( \text{IsSelectorIn}(s, d) \) that checks whether the selector \( s \) occurs in the selector list of the data structure \( d \). Note that in case a loop is involved these transformation rules all match on arbitrary \text{Iteration} terms. If no define-define and define-use dependencies exist between different body-instances the loop may be replaced by a \text{Forall} while still preserving the semantics. The so defined translation rules also cover \text{Forall} terms. Now consider the following example of a matrix multiplication (based on the example of [63]).
\[ AddMul(assignment \ z \ (x, +, (u, *, v)) \ s) \triangleq \]
\[ \text{library [addmul, } z, x, u, v] \ s \]

\[ SSPO(\text{iteration } [(i,n)] \text{ statements [library } [\text{addmul, } z, z, u, v] \ s11 ] \ s1) \triangleq \]
\[ \text{library [sspo, } i, n, z, u, v] \ s1 \]
\[ \text{if } \text{IsSelectorIn}(i, u) \text{ and } \text{IsSelectorIn}(i, v) \]

\[ VMO(\text{iteration } [(i,n)] \text{ statements [library } [\text{sspo}, j, m, z, u, v] \ s11 ] \ s1) \triangleq \]
\[ \text{library [vmo, } i, n, j, m, z, u, v] \ s1 \]
\[ \text{if } \text{IsSelectorIn}(i, z) \text{ and } \text{IsSelectorIn}(i, u) \]
\[ \text{and } \text{IsSelectorIn}(i, v) \text{ and } \text{IsSelectorIn}(j, v) \]
\[ \text{and not } \text{IsSelectorIn}(j, z) \text{ and not } \text{IsSelectorIn}(j, u) \]

\[ VM(\text{library } [\text{vinit, } i, n, z, c] \ s1, \text{library } [\text{vmo, } i, n, j, m, z, u, v] \ s2) \triangleq \]
\[ \text{library [vm, } i, n, j, m, z, u, v, c] \ s \]

\[ MM(\text{iteration } [(i,n)] \text{ statements [library } [\text{vm, } j, m, k, r, z, u, v, c] \ s11 ] \ s1) \triangleq \]
\[ \text{library [mm, } i, n, j, m, k, r, z, u, v, c] \ s1 \]
\[ \text{if } \text{IsSelectorIn}(i, z) \text{ and } \text{IsSelectorIn}(i, u) \]

Figure 7.1: Transformations of the Paramat library, converted to V-cal rules.
The names of the V-cal rules correspond to the names of the Paramat transformations.

iteration [(i,n)] statements [
    iteration [(j,m)] statements [
        assignment (C, [i,j]) 0 s111
    ] s11,
    iteration [(j,m)] statements [
        iteration [(k,r)] statements [
            assignment (C, [i,j]) ((C, [i,j]).+((A, [i,k]),*(B, [k,j]))) s1211
        ] s121
    ] s12
] s1

According to [63] the pattern recognition algorithm uses a leftmost depth first search, which can be coded into a strategy in our case. The pattern hierarchy graph can also be represented in the strategy. Applying the above presented transformation rules according to the strategy will first replace statement s111 by library [sinit, (C, [i,j]), 0]. The loop s11 with the sinit statement in its body then matches the Vinit rule such that until now the program is transformed to:
iteration [(i,n)] statements [  
  library [vinit, j, m, (C, [i,j]), 0] s11,  
  iteration [(j,m)] statements [  
    iteration [(k,r)] statements [  
      assignment (C, [i,j]) ((C, [i,j]), +.,((A, [i,k]), *.,(B, [k,j]))) s121  
    ] s121  
  ] s12  
] s1

Now we continue with matching the AddMul rule for statement s1211 such that it is replaced by library [addmul, (C, [i,j]), (C, [i,j]), (A, [i,k]), (B, [k,j])]. The loop s121 now matches with the SSPO rule such that the V-nus program looks like:

iteration [(i,n)] statements [  
  library [vinit, j, m, (C, [i,j]), 0] s11,  
  iteration [(j,m)] statements [  
    library [sspo, k, r, (C, [i,j]), (A, [i,k]), (B, [k,j])] s121  
  ] s12  
] s1

The loop s12 around the sspo statement can be transformed by the VMO rule to the library entry library [vmo, j, m, k, r, (C, [i,j]), (A, [i,k]), (B, [k,j])]. Together with the initialization statement s11 we may map this to a vm entry such that the program has become a single loop:

iteration [(i,n)] statements [  
  library [vm, j, m, k, r, (C, [i,j]), (A, [i,k]), (B, [k,j]), 0] s
] s1

This will in turn be matched by the MM rule. In this way we end up with a single library entry, representing the original matrix multiplication, in the following way:

library [mm, i, n, j, m, k, r, (C, [i,j]), (A, [i,k]), (B, [k,j]), 0] s1

With this example we showed that we can use existing libraries with transformation rules in our framework. In this way we will be able to have the same order of matches for numerical application programs as we would have with the original library.

7.7 Discussion

One goal in the design of V-cal was to provide a framework that allows compiler writers to construct compilers using a modular construction set technique. In this chapter we have demonstrated the use of our calculus based approach by presenting the implementation of a set of rules of an existing library. This case study shows that V-cal provides a suitable framework to express pattern matched program transformations.
7.7. DISCUSSION

One interesting question in the implementation of a V-cal engine is to determine in which order the permissible rules should be applied to a program. In many cases it is possible to compute this order automatically. However, sometimes it is desirable to have an explicit control on the evaluation order of the rules to achieve an optimal effect. In [95], [97], and [96] a detailed study is presented on the application of transformation rules for optimization. A framework is given that is based on an axiomatic specification technique and includes both pre-conditions and post conditions that must exist before and after applying optimizations. A conclusion of this work is that it is often ambiguous whether a certain transformation rule causes an optimization for the complete program in question. Despite computations for partial optimizations it seems that it is necessary to compose a strategy by hand for optimizing the complete program, or automatically try every possible transformation sequence. In general, this last option requires too many computation resources.
Chapter 8

Discussion

Hey Pal! How do I get to town from here? And he said: Well just take a right where they’re going to build that new shopping mall, go straight past where they’re going to put in the freeway, take a left at what’s going to be the new sports center, and keep going until you hit the place where they’re thinking of building that drive-in bank. You can’t miss it. And I said: This must be the place.

_Big Science_
Laurie Anderson

8.1 Conclusions

This thesis presents the tight relationship between compiler construction considerations and considerations on building an accompanying sound semantic framework. The cooperation of these two fields, in our case, mainly depends on the intermediate language _V-nus_. Instead of first constructing such a language, and then defining semantics for it, we worked the other way around. We started with a set of semantic features an intermediate language should have, after which we defined our language _V-nus_. Of course, in some cases specifying clear semantics was contrary to efficiency requirements (see Chapter 5: the _Forall_ term). However, with both fields in mind we were able to end up with a language that does no violence to either fields.

A major advantage of _V-nus_ for using it as an intermediate language for data parallel languages is the existence of multiple levels of abstraction. Data parallel languages often differ in the kind of language constructs that express parallelism (or more abstractly: cause parallelism) (see Chapter 1). By offering different levels of abstraction in _V-nus_, one can choose the abstraction level that is most appropriate when translating high-level data parallel programming languages to _V-nus_. Furthermore, we aimed at an optimizing compiler, in which multiple levels of abstraction are useful for expressing transformations. These transformations use the same language (_V-nus_) for both input and output. Apart
from optimizing transformations, other transformations are used to render V-nus programs from data parallelism to explicit parallelism. Together with the semantic framework one can verify whether these transformations preserve the original semantics.

Since the semantic framework was, among other reasons, necessary for transformation rule verification, we have used semantics based on an input/output relation. That is, no execution traces are part of the semantics, only the 'state' before and after execution of a V-nus term is deduced in this framework. In our experience, it suffices for the kind of compiler we are looking at.

Applying transformations to V-nus programs is taken care of by a separate 'engine' (see Chapter 1 and 7). As such, one can easily experiment with different sets of transformations and different techniques for application in order to compare the so compiled programs.

V-nus is indeed suited to serve as a platform for data parallel languages. A common language construct in data parallel languages is the forall term, and yet it has different semantics in different languages. It is a characteristic term for expressing independent computations, which, in turn, are suited for parallel computations. We have shown that the V-nus Forall term can be used to express a wide class forall semantics, and can, in our view, still be implemented efficiently. Data distribution declarations – another typical feature of data parallel languages – can be recorded in the V-nus symbol table.

V-nus is a self contained language; it is the only information that is exchanged between the different tools of the compiler. By necessity, the syntax of the language is extremely simple in order to be used efficiently by these tools. As a result, V-nus is not easily parsed by a human. Therefore, a V-nus related language is constructed, called Sugar V-nus, which syntax very much resembles the notations used in high-level data parallel languages. Currently, we have a compiler for V-nus, and an accompanying front-end that reads Sugar V-nus. In [28] we have shown that elemental parts of the Booster language can also be translated to V-nus. This is done by making use of rewrite rules, suited for the construction of a Booster front-end.

So, finally, we can conclude this thesis with answering 'yes' to our research question in Section 1.2.

### 8.2 Future work

One of the advantages of V-nus is the existence of multiple levels of abstraction, and, as such, offering a wide context for program transformations. In Section 1.8 we referred to [17] for further reading on this subject. It describes a 'rule based compiler' for V-nus, and presents a large set of transformations. At the time of writing this thesis, the semantics preserving requirement for these transformations has not been demonstrated by using our semantic model. Further research on this subject (i.e. the verification of transformations) should focus on the development of a generic transformation verification mechanism. Currently, transformations that affect only small parts of a program cause a rather extensive proof of their correctness. Such a generic verification mechanism should offer more proving tools (within the semantic model) such that more complex transformations can be handled
as well.

The observation that \textit{V-nus} is suited as an intermediate formalism is based on investigating general language concepts of different high-level data parallel languages. This assertion can be better founded by actually constructing front-ends for these high-level languages to \textit{V-nus}. Then, nested arrays need more attention in \textit{V-nus}. In general, operations on nested arrays cannot be translated to \textit{V-nus} directly; currently, views are needed as an intermediate data structure to cope with elements of nested arrays.

Incorporating nested arrays into \textit{V-nus} requires a thorough look at the data accessing expressions in the syntax of \textit{V-nus}, and a considerable change in the semantic model.

Currently, \textit{V-nus} is used for the construction of a compiler for the language SPAR [82]. SPAR is a Java related language with extensions for data parallel declarations.
Appendix A

The \textit{V-nus} grammar

We define the syntax of \textit{V-nus} using an extended BNF. The extensions have the following meaning:

\begin{center}
\begin{tabular}{ll}
\textit{EBNF Construct} & \textit{Meaning} \\
"\alpha" & \alpha \text{ is a terminal symbol} \\
[\alpha] & \text{zero or one } \alpha \\
(\alpha) & \alpha \text{ is a syntactical unit} \\
\alpha / / \beta & \text{a (possibly empty) sequence of } \alpha \text{’s separated by } \beta \text{’s} \\
\alpha^* & \text{an arbitrary number of } \alpha \text{’s} \\
\alpha^+ & \text{one or more } \alpha \text{’s} \\
\alpha \beta & \text{first } \alpha \text{ then } \beta \\
\alpha \mid \beta & \text{either } \alpha \text{ or } \beta \\
\end{tabular}
\end{center}
A.1 Units

Production ::= "program" Initializations Declarations Statements SymbolTable

Initializations ::= "initializations" "[" Initialization "/"," "]"

Initialization ::= Statements

Declarations ::= "declarations" "[" Declaration "/"," "]"

Declaration ::= Function
               | Procedure

Statements ::= "statements" [Label] "[" Statement "/"," "]"

Statement ::= [Pragmas] Action [Label]

Function ::= "function" IDENTIFIER FormalParameters IDENTIFIER Statements

Procedure ::= "procedure" IDENTIFIER FormalParameters Statements

Action ::= Control
         | Communication
         | Assignment
         | ViewStatement
         | ProcedureCall
         | LibraryEntry
         | Print

A.2 Declarations

FormalParameters ::= "[" FormalParameter "/"," "]"

FormalParameter ::= IDENTIFIER

Cardinalities ::= "[" Cardinality "/"," "]"

Cardinality ::= "(" IDENTIFIER "," Expression ")"
A.3 Flow of Control

Control ::= While
  | Iteration
  | Forall
  | Foreach
  | If
  | Statements
.
While ::= "while" Expression Statements
.
Iteration ::= "iteration" Cardinalities Statements
.
Forall ::= "forall" Cardinalities Statements
.
Foreach ::= "foreach" Cardinalities Statements
.
If ::= "if" Expression Statements Statements
.

A.4 Communication

Communication ::= Want
  | Synchronize
  | Redistribute
  | Barrier
  | Send
  | Receive
  | Broadcast
.
Barrier ::= "barrier"
.
Send ::= "send" Expression Expression Expression Expression
.
Receive ::= "receive" Expression Expression Expression Expression
.
Broadcast ::= "broadcast" Expression Expression
.
Want ::= "want" Label Expression
.
Synchronize ::= "synchronize" Label Expression
.
Redistribute ::= "redistribute" IDENTIFIER
A.5 Statements

\[
\text{Assignment} ::= \text{"assignment" Selection Expression} \\
| \quad \text{"assignment" FunctionCall Expression} \\

\text{ViewStatement} ::= \text{"view" IDENTIFIER View} \\

\text{ProcedureCall} ::= \text{"procedurecall" IDENTIFIER ActualParameters} \\

\text{LibraryEntry} ::= \text{"library" Expressions} \\

\text{Print} ::= \text{"printf" ActualParameters}
\]

A.6 Expressions

\[
\text{Expression} ::= \text{NUMBER} \\
| \quad \text{STRING} \\
| \quad \text{IDENTIFIER} \\
| \quad \text{View} \\
| \quad \text{Shape} \\
| \quad \text{Selection} \\
| \quad \text{FunctionCall} \\
| \quad \text{Ownership} \\
| \quad \text{Reduce} \\
| \quad \text{"\(" UNARY \text{" ," Expression \text{" )"}} \\
| \quad \text{"\(" Expression \text{" ," BINARY \text{" ," Expression \text{" )"}} \\
| \quad \text{"blocksize" Expression Expression} \\
| \quad \text{"size" Expression Expression} \\
| \quad \text{Pragma} \text{ Expression}
\]
A.7 Symbol Table

SymbolTable := "symboltable" Entries

Entries := "[" Entry "]"

Entry := "(" IDENTIFIER "," Pragmas "," EntryData ")"

EntryData := GlobalVariableData
| LocalVariableData
| ViewFormalData
| FunctionData
| ProcedureData
| FormalData
| ReturnValueData
| CardinalityVariableData
GlobalVariableData ::= "globalvariable" Type
LocalVariableData ::= "localvariable" IDENTIFIER Type
ViewFormalData ::= "viewformal" IDENTIFIER IDENTIFIER NUMBER BaseType
FunctionData ::= "functiondata" FormalParameters IDENTIFIER Type
ProcedureData ::= "proceduredata" FormalParameters
FormalData ::= "formal" IDENTIFIER
ReturnVariableData ::= "returnvariable" IDENTIFIER Type
CardinalityVariableData ::= "cardinalityvariable"
Type ::= BaseType
| "view" Forms BaseType
| "shape" Forms Distributions BaseType
Forms ::= [" Form ", "]"
Distributions ::= [" Distribution ", "]"
Distribution ::= "block"
| "cyclic"
| "blockcyclic" NUMBER
| "replicated"
| "collapsed"
Form ::= "natural" NUMBER
| "shapeformal" IDENTIFIER
| "expression" Expression
| "don'tcare"

All program variables, denoted with an IDENTIFIER, are represented by an Entry in the SymbolTable. The Pragmas and EntryData list the program variable's properties. We clarify the idea behind some EntryData of an arbitrary program variable (hereafter referred to with 'the variable') which might not be obvious.

The IDENTIFIER in LocalVariableData refers to the function that defines the locality of the variable. In ViewFormalData one can find successively the name of the function of which the variable is an actual parameter, the variable's formal parameter name, its dimensionality, and the type of the elements the view is referring to. FunctionData and ProcedureData represent the function header and procedure header respectively. In FormalData and ReturnVariableData the IDENTIFIER represents the corresponding func-
A.8  MISCELLANEOUS

Label ::= IDENTIFIER

BaseType ::= "boolean"
| "integer"
| "natural"
| "real"
| "complex"
| "string"

Pragma ::= "pragma!" [ Pragma / "", ""]

Pragma ::= "flag" IDENTIFIER
| "value" IDENTIFIER Expression

A.9  Tokens

Unary ::= "NOT" | "+" | "-" | "\"

Binary ::= ">" | ">=" | "<" | "<>
| "=" | "<=" | "<" | "+" | ";" | "+" | "/" | "&" | ";" | "|" | "&&" | "OR" | "DIV" | "MOD"

Identifier ::= LETTER (LETTER | Digit)*

Letter ::= "A" ... "Z" | "a" ... "z"

Digit ::= "0" ... "9"

Number ::= Digit+ [ "." Digit+] ["E" | "+" | "-" ] Digit+

String ::= """" (Character | EscapedCharacter)* """

Character ::= \( \Sigma \setminus (\"\" | \"\")\)

EscapedCharacter ::= \"\" | \\f | \"\n" | \"\r" | \"\t" | "\" | "\"
| \"\" Digit Digit Digit]

\( \Sigma \) contains all printable ASCII characters " " ... "". The character sequence """" that appears on the right hand side of the grammar rule for STRING denotes the terminal double
A.10 White Space

White space outside strings may be used to separate tokens and is no part of the language. It should be filtered out before parsing starts under the condition that separated tokens remain separated. Beside space, tab, linefeed, carriage return, and formfeed comments also count as white space. There are two types of comments:

- A comment starts with "/" and ends at the end of the line. This is a single line comment.
- A comment starts with "/*" and ends with "*/". These comments may be nested.
Appendix B

Function definitions

In this appendix one can find the functions $M_\Sigma$, $M_\Gamma$, $M_\Psi$, $\mathcal{E}$, $Loc$, and $IP$, as these are defined in the Chapters 4, 5, and 6. These functions have been modified throughout the mentioned chapters; here, their complete definitions are listed.

B.1 The meaning function $M_\Sigma$

The meaning function $M_\Sigma : v_{\text{num}} \to \Delta \to \Phi \to \Sigma \to \Sigma$ of Chapter 4 and Chapter 5 (where it was simply called $M$), and of Chapter 6, is completely defined by the following definitions.

1. $M(\text{ViewStatement})(\delta)(\varphi)(\sigma) = \sigma \triangleleft \sigma'_V$
   
   where
   
   $\begin{align*}
   \text{ViewStatement} &= \text{view Id View} \\
   \sigma'_V &= \{(\text{Id}, s, ip)\} \\
   ip &= IP(\text{View})(\delta)(\varphi)(\sigma) \\
   s &= \sigma_L(\ell)
   \end{align*}$

2. $M(\text{Assignment})(\delta)(\varphi)(\sigma) = \sigma \triangleleft \sigma'_L$
   
   where
   
   $\begin{align*}
   \text{Assignment} &= \text{assignment Selection Expression} \\
   \sigma'_L &= \{(\text{Loc}(\text{Selection})(\delta)(\varphi)(\sigma), \mathcal{E}(\text{Expression})(\delta)(\varphi)(\sigma))\}
   \end{align*}$

3. $M(\text{procedurecall } Id_p [e_0, \ldots, e_n])(\delta)(\varphi)(\sigma) = \sigma'$
   
   where
   
   $\begin{align*}
   \sigma_0 &= \text{NewScope}(\sigma) \\
   \sigma_1 &= \sigma_0 \triangleleft \text{Pass}(p_0, e_0, t_0)(\delta)(\varphi)(\sigma_0) \triangleleft \ldots \\
   &\triangleleft \text{Pass}(p_n, e_n, t_n)(\delta)(\varphi)(\sigma_0) \\
   \sigma_2 &= M(\text{Statements})(\delta)(\varphi)(\sigma_1) \\
   \sigma' &= \text{LeaveScope}(\sigma_2) \\
   s &= \sigma_{0_n}(\ell) \\
   [p_0, \ldots, p_n] &= \delta(Id_p, s, \text{par}) \\
   [t_0, \ldots, t_n] &= \delta(Id_p, s, \text{partype}) \\
   \text{Statements} &= \delta(Id_p, s, \text{stm})
   \end{align*}$
4. \( M(\text{statements } [c]) = M(c) \)

5. \( M(c_1, c_2)(\delta)(\varphi) = M(c_2)(\delta)(\varphi) \circ M(c_1)(\delta)(\varphi) \)

6. \( M(\text{If})(\delta)(\varphi)(\sigma) = \begin{cases} M(\text{Statements}_1)(\delta)(\varphi)(\sigma) & \text{if } B(\text{Expression})(\delta)(\varphi)(\sigma) \\ M(\text{Statements}_2)(\delta)(\varphi)(\sigma) & \text{otherwise} \end{cases} \)
   
   where \( \text{If} = \text{if } \text{Expression } \text{Statements}_1 \text{ Statements}_2 \)

7. \( M(\text{While}) = \text{lub}_{\text{gen}} CL_i(\text{While}) \)

8. \( M(\text{Iteration})(\delta)(\varphi)(\sigma) = SL(\text{Statements})(T)(\delta)(\varphi)(\sigma) \)
   
   where \( \begin{cases} \text{Iteration} = \text{iteration Cardinalities Statements} \\ T = DP(\text{Cardinalities})(\delta)(\varphi)(\sigma) \end{cases} \)

9. \( M(\text{Forall})(\delta)(\varphi)(\sigma) = \mathcal{P}(\text{Statements})(T)(\delta)(\varphi)(\sigma) \)
   
   where \( \begin{cases} \text{Forall} = \text{forall Cardinalities Statements} \\ T = DP(\text{Cardinalities})(\delta)(\varphi)(\sigma) \end{cases} \)

10. \( M(\text{Foreach})(\delta)(\varphi)(\sigma) = SL'(\text{Statements})(T)(\delta)(\varphi)(\sigma) \)
    
    where \( \begin{cases} \text{Foreach} = \text{foreach Cardinalities Statements} \\ T = DP(\text{Cardinalities})(\delta)(\varphi)(\sigma) \end{cases} \)

\section*{B.2 The meaning function \( M_\Gamma \)}

The meaning function \( M_\Gamma : \text{Vnus} \to \Gamma \to \Psi \) of Chapter 6 is completely defined by the following definitions. Remember that \( PS \) is the set of singular \( \text{V-nus} \) terms for which \( M_\Sigma \) (above) is defined.

1. \( M_\Gamma(c)(\gamma) = \{ (\delta, \varphi, \sigma', \gamma_B) \} \) if \( c \in PS \)
   
   where \( \begin{cases} \gamma = (\delta, \varphi, \sigma, \gamma_n) \\ \sigma' = M_\Sigma(c)(\delta)(\varphi)(\sigma) \end{cases} \)

2. \( M_\Gamma(\text{Send})(\gamma) = \begin{cases} \{ \gamma \} & \text{if } \sigma_{L}(\varphi) \neq \bar{f} \\ \{ (\delta, \varphi, \sigma, \gamma'_B) \} & \text{otherwise} \end{cases} \)
   
   where \( \begin{cases} \text{Send} = \text{send } f \, t \, e \\ \gamma = (\delta, \varphi, \sigma, \gamma_B) \\ \gamma'_B = \gamma_B \bullet (\bar{f}, \bar{t}, \bar{e}) \\ \bar{f} = \mathcal{E}(f)(\delta)(\varphi)(\sigma) \\ \bar{t} = \mathcal{E}(t)(\delta)(\varphi)(\sigma) \\ \bar{e} = \mathcal{E}(e)(\delta)(\varphi)(\sigma) \end{cases} \)
B.3. THE MEANING FUNCTION \( \mathcal{M}_\Psi \)

3. \( \mathcal{M}_\Gamma(\text{Receive})(\gamma) = \begin{cases} \{\gamma\} & \text{if } \sigma_L(\varphi) \neq \tilde{t} \\ X & \text{otherwise} \end{cases} \)

\begin{align*}
\text{Receive} &= \text{receive } t \text{ } f \text{ } Id \\
X &= \{\gamma' \mid \gamma' = (\delta, \varphi, \sigma', \gamma_B), \gamma'_B = \gamma_B \bullet (\tilde{f}, \tilde{t}, d), d \in \text{Data}\} \\
\gamma &= (\delta, \varphi, \sigma, \gamma_B) \\
\tilde{f} &= \mathcal{E}(f)(\delta)(\varphi)(\sigma) \\
\tilde{t} &= \mathcal{E}(t)(\delta)(\varphi)(\sigma) \\
\sigma' &= \text{Assign}(d, Id)(\sigma)
\end{align*}

4. \( \mathcal{M}_\Gamma(c_1, c_2)(\gamma) = \bigcup_{\gamma' \in \mathcal{M}_\Gamma(\gamma)} \mathcal{M}_\Gamma(c_2)(\gamma') \)

B.3 The meaning function \( \mathcal{M}_\Psi \)

The meaning function \( \mathcal{M}_\Psi : \text{Vnum} \rightarrow \Phi \rightarrow \Psi \) of Chapter 6 is completely defined by the following definition.

\[ \mathcal{M}_\Psi(V)(\psi) = \left[ \bigcup_{\gamma \in \psi} \mathcal{M}_\Gamma(V)(\gamma) \right]_{\Psi} \]

B.4 The semantic function \( \mathcal{E} \)

The semantic function \( \mathcal{E} : \mathcal{L}(\text{Expression}) \rightarrow \Delta \rightarrow \Phi \rightarrow \Sigma \rightarrow \text{Data} \) of Chapter 4 and Chapter 5 is completely defined by the following definitions.

Let \( \otimes \in \{+, -, \times, \div, \text{MOD}\} \) represent a binary operator of the V-num language, but not the operator \( ! \). Let \( \in \in \{>, >=, <>, =, <=, <, \text{AND}, \text{OR}\} \) represent a relational operator of the V-num language.

1. \( \mathcal{E}(d)(\delta)(\varphi)(\sigma) = d \) if \( d \in \text{Data}_0 \)
2. \( \mathcal{E}(i)(\delta)(\varphi)(\sigma) = \varphi(i) \) if \( i \in I \)
3. \( \mathcal{E}(\text{shape } [c_1, \ldots, c_n])(\delta)(\varphi)(\sigma) = (\mathcal{E}(c_1)(\delta)(\varphi)(\sigma), \ldots, \mathcal{E}(c_n)(\delta)(\varphi)(\sigma)) \)
4. \( \mathcal{E}((i, 1, c))(\delta)(\varphi)(\sigma) = \text{index}(\mathcal{E}(i)(\delta)(\varphi)(\sigma), \text{Sel}(c)(\delta)(\varphi)(\sigma)) \)
5. \( \mathcal{E}((c_1, \otimes, c_2)) = \mathcal{E}(c_1) \otimes \mathcal{E}(c_2) \)
6. \( \mathcal{E}((c_1, \sqsubseteq, c_2)) = \mathcal{B}(c_1, \sqsubseteq, c_2) \)
7. \( \mathcal{E}([\text{true}]) = \mathcal{B}(\text{true}) \)
8. \( \mathcal{E}([\text{false}]) = \mathcal{B}(\text{false}) \)
9. \( \mathcal{E}(\text{View})(\delta)(\varphi)(\sigma) = \text{ip2data}(\text{ip})(\sigma) \)

where \( \text{ip} = \text{IP}(\text{View})(\delta)(\varphi)(\sigma) \)
10. \( \mathcal{E}(\text{Id})(\varphi)(\sigma) = \text{ip2data}(\text{ip})(\sigma) \) if \( \text{Id} \in V \)
where 
   \[
   \begin{align*}
   \text{ip} &= \sigma_V(\text{Id}, s) \\
   s &= \sigma_L(\ell)
   \end{align*}
   \]

11. \( \mathcal{E}(\text{FunctionCall})(\delta)(\varphi)(\sigma) = \text{ip2data}(\text{ip})(\sigma) \)
where \( \text{ip} = \mathcal{I}(\text{FunctionCall})(\delta)(\varphi)(\sigma) \)

12. \( \mathcal{E}(\text{(Expression, Selectors)})(\delta)(\varphi)(\sigma) = d_i \)
where 
   \[
   \begin{align*}
   d &= \mathcal{E}(\text{Expression})(\delta)(\varphi)(\sigma) \\
   i &= \mathcal{I}(\text{Selectors})(\delta)(\varphi)(\sigma)
   \end{align*}
   \]

13. \( \mathcal{E}(\text{Reduce})(\delta)(\varphi)(\sigma) = \mathcal{E}(\text{Structure})(\delta)(\varphi \bowtie \varphi_0)(\sigma) \bowtie \ldots \)
\( \bowtie \mathcal{E}(\text{Structure})(\delta)(\varphi \bowtie \varphi_{n-1})(\sigma) \)
\( \text{Reduce} = \text{reduce} \bowtie (\text{Cardinalities, Structure}) \)
where 
   \[
   \begin{align*}
   n &= |\mathcal{D}(\text{Cardinalities})(\delta)(\varphi)(\sigma)| \\
   \varphi_0, \ldots, \varphi_{n-1} &\in \mathcal{D}(\text{Cardinalities})(\delta)(\varphi)(\sigma)
   \end{align*}
   \]

B.5 The semantic function \( \text{Loc} \)

The function \( \text{Loc}: \mathcal{L}(\text{Selection}) \to \Delta \to \Phi \to \Sigma \to \text{Loc} \) of Chapter 4 is completely defined by the following definitions.

1. \( \text{Loc}(\text{Id, Selectors})(\delta)(\varphi)(\sigma) = \sigma_V(\text{Id}, s)(i) \)
where 
   \[
   \begin{align*}
   s &= \sigma_L(\ell) \\
   i &= \mathcal{I}(\text{Selectors})(\delta)(\varphi)(\sigma)
   \end{align*}
   \]

2. \( \text{Loc}(\text{FunctionCall, Selectors})(\delta)(\varphi)(\sigma) = \text{ip}(i) \)
where 
   \[
   \begin{align*}
   \text{ip} &= \mathcal{I}(\text{FunctionCall})(\delta)(\varphi)(\sigma) \\
   i &= \mathcal{I}(\text{Selectors})(\delta)(\varphi)(\sigma)
   \end{align*}
   \]

B.6 The semantic function \( \mathcal{I}(\text{P}) \)

The function \( \mathcal{I}(\text{P}): \mathcal{L}(\text{Expression}) \to \Delta \to \Phi \to \Sigma \to \text{IPS} \) of Chapter 4 is completely defined by the following definitions.

Let \( \text{Cardinalities} \) be a cardinality list for the index identifiers \( i_0, \ldots, i_n \).

1. \( \mathcal{I}(\text{P})(\text{Cardinalities, Selection})(\delta)(\varphi)(\sigma) = \)
\( \{ (\varphi'(i_0), \ldots, \varphi'(i_n)), \text{Loc}(\text{Selection})(\delta)(\varphi \bowtie \varphi')(\sigma) \} \)
\( \varphi' \in \mathcal{D}(\text{Cardinalities})(\delta)(\varphi)(\sigma) \} \)
2. $\text{IP}(\text{functioncall} ~ Id_F \ [e_0, \ldots, e_n]) (\delta)(\varphi)(\sigma) = ip$

$\sigma_0 = \text{NewScope}(\sigma)$

$\sigma_1 = \sigma_0 \triangleleft \text{Pass}(p_0, e_0, t_0)(\delta)(\varphi)(\sigma_0) \triangleleft \ldots$

$\triangleleft \text{Pass}(p_n, e_n, t_n)(\delta)(\varphi)(\sigma_0)$

$\sigma' = \mathcal{M}(\text{Statements})(\delta)(\emptyset)(\sigma_1)$

$ip = \sigma'_V(Id_R, s)$

$s = \sigma_{0L}(\ell)$

$p_0, \ldots, p_n = \delta(Id_F, s, \text{par})$

$t_0, \ldots, t_n = \delta(Id_F, s, \text{par type})$

Statements = $\delta(Id_F, s, \text{stm})$

$Id_R = \delta(Id_F, s, \text{result})$
Appendix C

Examples of forall statements

An overview is presented of examples of forall statements as can be found in the languages CM Fortran, HPF, and V-nus. Syntax and semantics are not described in general, but the differences and correspondences of these forall statements are shown through a set of examples. The examples are presented in a pseudo language. Translating the pseudo language to CM Fortran, HPF, and V-nus is straightforward.

The following assumptions are made for the ease of presenting the examples:

- It is assumed that all left hand sides are conformable with their right hand sides.
- Equally named variables of different examples may have nothing in common.
- Differently named variables are independent of each other, unless stated otherwise.

---

Example C.1 Scalar assignment.

```fortran
forall i=0,n
   a[i] := expr
end
```

`expr` is an arbitrary expression that represents a scalar and may depend on `i`.

---

CM Fortran The value of `expr` is computed for all `i`. Then all `a[i]` become the value of `expr` in parallel.

HPF and V-nus The value of `expr` is computed for all `i`. Then all `a[i]` become the value of `expr` in arbitrary order.
Example C.2  *Static array assignment.*

\[
\text{forall } i=0,n \\
\quad a[i,p..q] := b[r..s] \\
\text{end}
\]

\(p,\, q,\, r,\, \text{and } s\) are arbitrary scalars (but \(p-q = r-s\) yields).

**CM Fortran**  All arrays \(a[i,p..q]\) become the array \(b[r..s]\) in parallel.

**HPF**  All arrays \(a[i,p..q]\) become the array \(b[r..s]\) in arbitrary order.

**V-nus**  Not valid. To express the same assignment, this example will be translated to:

\[
\text{forall } i=0,n \quad j=0,m \\
\quad a[i,p+j] := b[r+j] \\
\text{end}
\]

where \(m = p-q\). See example C.8 for the meaning of this *forall* statement.

Example C.3  *Dynamic array assignment.*

\[
\text{forall } i=0,n \\
\quad a[i,p..q] := b[i..i+x] \\
\text{end}
\]

\(p,\, q,\, \text{and } x\) are scalars.

**CM Fortran**  Each array \(a[i,p..q]\) becomes the array \(b[i..i+x]\). The assignments are executed sequentially. Remark: the right hand side expression is computed first, then the
assignments take place. If a data dependency exists between a and b, it still does not matter which execution order is chosen.

**HPF** All arrays a[i,p,q] become the array b[i..i+x] in arbitrary order.

**V-nus** Not valid. To express the same assignment, this example will be translated to:

```plaintext
forall i:=0,n j:=0,m
  a[i,p+j] := b[i+j]
end
```

Note: in *Booster* it is not allowed to use the index identifier in expressions representing array bounds.

---

**Example C.4** A function call.

```plaintext
forall i:=0,n
  a[i] := f(X)
end
```

F is a function and X represents the actual parameter list.

---

**CM Fortran** All a[i] become the value of f(X) sequentially. It depends on the ‘intelligence’ of the implementation how often the function call f(X) is executed. Note, that this has no influence on the meaning of this forall statement.

**HPF** The same semantics yield as for CM Fortran, except that the assignment may be performed in arbitrary order. In this case F must be a pure function as is defined in the language specification of HPF.

**V-nus** The same semantics yield as for HPF, extended by the requirement that F must be a referentially transparent function.
Example C.5  *Multiple assignments.*

```fortran
forall i=0,n
  a[i] := expr_1
  b[i] := expr_2
end
```

**CM Fortran**  Not valid. Only one assignment may occur in the body of a `forall` statement. Therefore, also in Example C.6 and C.7 no semantics are presented for CM Fortran.

**HPF**  The semantics are equal to the following program fragment:

```fortran
forall i=0,n a[i] := expr_1 end;
forall i=0,n b[i] := expr_2 end
```

where all expressions that are present in the original `forall` statement are fixed before it is split into two `forall` statements.

**V-nus**  Each body instance is executed sequentially. But the order in which the body instances are chosen is arbitrary.

Example C.6  *Multiple assignments with at least one function call.*

```fortran
forall i=0,n
  a[i] := expr
  b[i] := f(X)
end
```

**HPF**  The semantics follow from Example C.4 and C.5.
**V-nus**  Here too, the semantics follow from Example C.4 and C.5. The function call will be executed \( n \) times.

---

**Example C.7** Multiple assignments with a data dependency.

\[
\text{forall } i=0,n \\
\quad a[i] := expr_1 \\
\quad a[i+1] := expr_2 \\
\text{end}
\]

---

**HPF**  The semantics follow from Example C.5.

**V-nus**  Not ‘valid”; it is a non-deterministic forall. A define-define dependency exists for \( a[i], 1 \leq i \leq n - 2 \). It is forbidden to have a define-define dependency between two different body instances. Define-define dependencies are allowed if they occur in the same body instance. For example, if the subscript \( i+1 \) is replaced by \( i \) then the forall statement has become valid.

---

**Example C.8** Multiple index variables.

\[
\text{forall } i=0,n \ j=0,m \\
\quad a[i,j] := expr \\
\text{end}
\]

---

**CM Fortran**  The total index space \( I \) is the cartesian product: \([0 \ldots n-1] \times [0 \ldots m-1]\). All elements \( a[i,j] \) become the value \( expr \) in parallel, for \((i,j) \in I\).

**HPF and V-nus**  The same semantics yield as for CM Fortran, except that the assignments may be done in arbitrary order. Furthermore, it is not allowed that the \( n \)-expression and the \( m \)-expression are defined in terms of \( i \) and \( j \).
Example C.9 A nested forall statement.

forall i=0,n
    forall j=0,m
       a[i,j] := expr
    end
end

CM Fortran Not valid. Only so-called ‘forall-assignments’ are allowed in the forall body.

HPF and V-nus As in Example C.8 an index space I is created which is the set of subscripts: $[0 \ldots n-1] \times [0 \ldots m-1]$. All subscriptable elements become the value of \textit{expr} in arbitrary order. However, now it is allowed to define the m-expression in terms of i.
Appendix D

Transformation rule verification

The transformation rules and verifications of their correctness presented in this appendix are not presented in full detail. An outline is given of how a proof of correctness can be given by using the semantic model presented in this thesis.

D.1 Construct substitution

Consider:

\[
CS(\text{iteration } \text{Cardinalities } \text{Statements}) \triangleq \\
\begin{align*}
\text{forall } \text{Cardinalities } \text{Statements} \\
\text{if not } DD(\text{Cardinalities}, \text{Statements})(ddi) \\
\text{and not } DU(\text{Cardinalities}, \text{Statements})(ddi)
\end{align*}
\]

where \( ddi \) is the data dependency information for the program to be considered. Its correctness is proved in the context of meanings in the meaning function space: \( \Sigma \rightarrow \Sigma \). For ease of reading the transformation rule to be verified is simplified by not considering \( \text{Cardinalities} \) in general.

Proof Assume that \( \delta \in \Delta, \varphi \in \Phi, \sigma \in \Sigma \) are valid states for the program to which the transformation rule is applied. Then by Definition B.1 and Definition 4.7.6 it can be deduced that:

\[
\begin{align*}
M_{\Sigma}(\text{iteration } [(j,n)]) \text{Statements}(\delta)(\varphi)(\sigma) &= \\
SL(\text{Statements})(DP)(\delta)(\varphi)(\sigma) &= \\
\sigma_n
\end{align*}
\]

where

\[
\begin{align*}
\sigma_i &= M_{\Sigma}(\text{Statements})(\delta)(\varphi_i)(\sigma_{i-1}) \quad 1 \leq i \leq n \\
\sigma_0 &= M_{\Sigma}(\text{Statements})(\delta)(\varphi_0)(\sigma) \\
\varphi_i &= \varphi \triangleleft (j, i) \quad 0 \leq i \leq n \\
DP &= DP([(j,n)])(\delta)(\varphi)(\sigma)
\end{align*}
\]

147
So
\[ \sigma_i = M_\Sigma(Statements)(\delta)(\varphi \land (j, i))(\sigma_{i-1}) \quad 1 \leq i \leq n \]
\[ \sigma_0 = M_\Sigma(Statements)(\delta)(\varphi \land (j, 0))(\sigma) \]
Since program states only 'grow', formalized in Theorem 5.1, and by Definition 3.6.1 for the replacement function \textless, we may write:
\[ \sigma_n = \sigma \textless \sigma_0 \textless \ldots \textless \sigma_n \]
We can as well 'update' \( \sigma \) with the changes in the \( \sigma_i \) relative to \( \sigma \) using Definition 5.4.2.
\[ \sigma_n = \sigma \textless \text{diff}_0 \textless \ldots \textless \text{diff}_n \]
where \( \text{diff}_i = \sigma_i \div \sigma \quad 0 \leq i \leq n \)
Since the \( \sigma_i \) steadily grow, the \( \text{diff}_i \) grow too (for ascending \( i \)). So, we look at 'the differences between the differences', formally specified as
\[ \text{diff}'_{i+1} = \text{diff}_{i+1} \div \text{diff}_i \quad 0 \leq i < n \]
\[ \text{diff}'_0 = \text{diff}_0 \]
Since there is no define-define dependency between \textit{Statements} for all \textit{Cardinalities} the \textit{diff}'_i are mutual exclusive, and can be reordered. Furthermore, since no define-use dependency exists we can as well look at the meaning of \textit{Statements} in the program state \( \sigma \) for all \textit{Cardinalities}. Therefore we define:
\[ \forall i \in \{0, \ldots, n\}: \sigma'_i = M(Statements)(\delta)(\varphi_i)(\sigma) \]
\[ \forall i \in \{0, \ldots, n\}: \text{diff}''_i = \sigma_i \div \sigma \]
and we claim that \( \text{diff}''_i = \text{diff}'_i \) for all \( i \). This can easily be verified by Definition 5.4.2 for \( \div \).
So, we conclude:
\[ \sigma_n = \sigma \textless \text{diff}_0 \textless \ldots \textless \text{diff}_n = M_\Sigma(\forall [j, n] \textit{Statements})(\delta)(\varphi)(\sigma) \]
where \( (i_0, \ldots, i_n) \) is a predefined permutation of \((0, \ldots, n)\).

D.2 Communication Insertion

Consider:
\[
\text{CI} \text{assignment } l h s \text{ Expression } s \quad \hat{=} \\
\quad \text{want } s \text{ } D_1 \text{ s}_1 \quad \ldots \quad \text{want } s \text{ } D_n \text{ s}_n \,
\quad \text{assignment } l h s \text{ Expression } s \,.
\quad \text{synchronize } s \text{ } l h s \text{ s}_m
\]
where the data structures \( D_1, \ldots, D_n \) are used in \textit{Expression}, and \( s_1, \ldots, s_n, s_m \in FLAB \).
Here, \( lhs \) represents both the non-terminals \( Selection \) and \( FunctionCall \).

Again, for ease of reading we simplify this transformation rule by considering a scalar assignment, leaving us with:

\[
CI(assignment \; (x, \; []) \; (y, \; [])) \triangleq \\
\quad \text{want } s \; (y, \; []) \; s1, \\
\quad \text{assignment } (x, \; []) \; (y, \; []) \; s, \\
\quad \text{synchronize } s \; (x, \; []) \; s2
\]

Next we assume the use of the 'owner computes rule' which makes the \( Synchronize \) redundant. We also assume the correctness of the application of this owner computes rule. As such the transformation rule to be considered is evolved to:

\[
CI(assignment \; (x, \; []) \; (y, \; [])) \triangleq \\
\quad \text{want } s \; (y, \; []) \; s1, \\
\quad \text{if isowner } (x, \; []) \; \text{me statements } [ \\
\quad \quad \text{assignment } (x, \; []) \; (y, \; []) \; s \\
\quad \quad ] \; \text{statements } [ ]
\]

Finally, since we considered the \( Want \) to be a short-cut for a \( Send \) - \( Receive \) pair, we will focus on the following rule instance, where we assume that \( (x, \; []) \) resides on process \( p_x \) and \( (y, \; []) \) resides on process \( p_y \).

\[
CI(assignment \; (x, \; []) \; (y, \; [])) \triangleq \\
\quad \text{send } p_y \; p_x \; (y, \; []), \\
\quad \text{receive } p_x \; p_y \; (y, \; []), \\
\quad \text{if isowner } (x, \; []) \; \text{me statements } [ \\
\quad \quad \text{assignment } (x, \; []) \; (y, \; [])) \\
\quad \quad ] \; \text{statements } [ ]
\]

Note: for efficiency considerations one might draw the \( Send \) and \( Receive \) into the rightly parts of the \( If \). Let’s call \( V \) the program that is the result after applying the transformation rule.

For the verification we cannot require to have the same state changes for both parts of the transformation rule. Instead, we compare the values of the program variables \( x \) and \( y \) as given by their owners.

**Proof** Let \( \delta \in \Delta, \varphi \in \Phi, \sigma \in \Sigma \) be valid states for the program to which the rule is applied. Say \( \sigma_V(x, s)((() = x \) and \( \sigma_V(y, s)((() = y \) and \( s \) is the 'current scope'. Then, with Definition B.1:

\[
M_\Sigma(assignment \; (x, \; []) \; (y, \; [])) (\delta)(\varphi)(\sigma) = \sigma'
\]

such that \( \sigma'_V(x, s)((() = y \) and \( \sigma'_V(y, s)((() = y \)
APPENDIX D. TRANSFORMATION RULE VERIFICATION

For the rest of this proof we use the following convention: for an arbitrary process state \( \gamma \) we may refer to the states \( \delta, \varphi, \sigma, \gamma_B \) such that \( \gamma = (\delta, \varphi, \sigma, \gamma_B) \).

1. Consider process \( p_x \); say \( \gamma_x \in \Gamma \) represents \( p_x \) and is a valid process state for \( V \). Then, with Definition B.2:

\[
\mathcal{M}_\Gamma(\text{send } p_y \ p_x \ (y, [ ]))(\gamma_x) = \{\gamma_x\}
\]

\[
\mathcal{M}_\Gamma(\text{receive } p_x \ p_y \ (y, [ ]))(\gamma_x) = \{\gamma_x' | \sigma'_{xv}(y, s)(()) = d, \gamma'_{xB} = \gamma_{xB} \bullet (\overline{p_y}, \overline{p_x}, d), \ d \in \text{Data}\}
\]

The expression \( \mathcal{E}(\text{isowner } (x, [ ] ) \ me) \) evaluates to \text{true} for valid states in this context. Say \( IfStm \) is a shortcut for the complete \( If \) statement in the program fragment above. We gain:

\[
\mathcal{M}_\Gamma(\text{IfStm})(\gamma_x') = \{\gamma''_x\}
\]

\[
\mathcal{M}_\Gamma(\text{assignment } (x, [ ] ) \ (y, [ ]))(\gamma_x') = \{\gamma''_x\}
\]

where \( \sigma''_{xv}(x, s)(()) = \sigma''_{xv}(y, s)(()) = d \)

for all process states \( \gamma_x' \) as deduced above.

2. Consider process \( p_y \); say \( \gamma_y \in \Gamma \) represents \( p_y \) and is a valid process state for \( V \). Then, with Definition B.2:

\[
\mathcal{M}_\Gamma(\text{send } p_y \ p_x \ (y, [ ]))(\gamma_y) = \{\gamma_y\} \quad \text{where} \quad \gamma'_{yB} = \gamma_{yB} \bullet (\overline{p_y}, \overline{p_x}, y)
\]

\[
\mathcal{M}_\Gamma(\text{receive } p_x \ p_y \ (y, [ ]))(\gamma_y) = \{\gamma_y\}
\]

For process \( p_y \) the expression \( \mathcal{E}(\text{isowner } (x, [ ] ) \ me) \) evaluates to \text{false} in \( \gamma_y' \). Thus:

\[
\mathcal{M}_\Gamma(\text{IfStm})(\gamma_y') = \{\gamma_y'\}
\]

We are interested in the meaning of \( V \) expressed as \( \mathcal{M}_\Psi(V)(\{\gamma_x, \gamma_y\}) \). As all process states \( \gamma''_x \) and the process state \( \gamma''_y \) must completely match. Of course, only \( \gamma''_x \) with \( \gamma''_{xB} = \gamma_{xB} \bullet (p_y, p_x, y) \) matches with \( \gamma_y \). This particular \( \gamma''_x \) is called \( \gamma^*_x \). Then:

\[
\mathcal{M}_\Psi(V)(\{\gamma_x, \gamma_y\}) = \{\gamma^*_x, \gamma_y\}
\]

For the first part of the transformation rule the ‘owner’ of \( x \) sets \( x \) to \( y \) (i.e. \( \sigma_{xV}^+(x, s)(()) = y \)). For the second part of this transformation rule (i.e. \( V \)) the owner of \( x \) sets \( x \) to \( y \) as well (i.e. \( \sigma_{xV}^+(x, s)(()) = y \)). Since \( y \) is not changed, the owners of \( y \) (in both parts of the rule) leave the value of \( y \) untouched, and thus equal for both parts.

\[\square\]
**Appendix E**

**Table of symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnus</td>
<td>set of all V-nus language terms (p. 17)</td>
</tr>
<tr>
<td>Σ</td>
<td>set of program states (p. 34)</td>
</tr>
<tr>
<td>σ</td>
<td>a program state (p. 34)</td>
</tr>
<tr>
<td>Γ</td>
<td>set of process states (p. 100)</td>
</tr>
<tr>
<td>γ</td>
<td>a process state (p. 100)</td>
</tr>
<tr>
<td>Ψ</td>
<td>set of parallel program states (p. 100)</td>
</tr>
<tr>
<td>ψ</td>
<td>a parallel program state (p. 100)</td>
</tr>
<tr>
<td>⊗</td>
<td>math operator representing V-nus operator ⊗ (p. 17)</td>
</tr>
<tr>
<td>ℒ(N)</td>
<td>class of N (p. 17)</td>
</tr>
<tr>
<td>A_{(i,j)}</td>
<td>element of data structure A on coordinate (index) (i, j) (p. 29)</td>
</tr>
<tr>
<td>ℤ_{d,u}</td>
<td>bounded set with lower bound l, upper bound u, and dimension d (p. 31)</td>
</tr>
<tr>
<td>f : X → Y</td>
<td>function f of type X → Y (p. 29)</td>
</tr>
<tr>
<td>dom(f)</td>
<td>domain of f (p. 74)</td>
</tr>
<tr>
<td>f a g</td>
<td>replacement of g over (parts of) f (p. 38)</td>
</tr>
<tr>
<td>f ÷ g</td>
<td>difference of f with respect to g (p. 80)</td>
</tr>
<tr>
<td>γ ≈ γ'</td>
<td>(partial) match between process states γ and γ' (p. 104)</td>
</tr>
<tr>
<td>[ψ]_κ</td>
<td>complete match over parallel program state ψ (p. 105)</td>
</tr>
<tr>
<td>Λ</td>
<td>special location containing the current scope (p. 33)</td>
</tr>
<tr>
<td>φ</td>
<td>special location containing the current process number (p. 100)</td>
</tr>
<tr>
<td>ℙ(X)</td>
<td>powerset of X (p. 29)</td>
</tr>
<tr>
<td>X</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td>&lt;X</td>
<td>partial lexicographical order on strings of length x (p. 68)</td>
</tr>
<tr>
<td>&lt;α</td>
<td>lexicographical order (p. 68)</td>
</tr>
<tr>
<td>≤Ψ</td>
<td>order on ordered index states (p. 69)</td>
</tr>
<tr>
<td>=Φ</td>
<td>domain equivalence on ordered index states (p. 69)</td>
</tr>
<tr>
<td>minΦ</td>
<td>minimum of ordered index states (p. 70)</td>
</tr>
</tbody>
</table>
Bibliography


Samenvatting

Onderzoek in semantiek en vertalerbouw (compiler construction) is vaak gescheiden. De eerste heeft een theoretisch karakter, terwijl de tweede een meer toegepast karakter heeft. Echter, beide vakgebieden zijn van fundamenteel belang bij de interactie tussen een programmeertaal en het gedrag van een computer.

De semantiek van een programmeertaal geeft een betekenis aan ‘expressies’ die met die taal gemaakt kunnen worden. Een ‘vertaler’ (compiler) vertaalt programma’s in een programmeertaal naar programma’s in een andere taal op een meestal lager abstractie niveau. Natuurlijk moet de originele betekenis van een programma behouden blijven gedurende het compilatie proces. Er is dus een hechte band tussen semantiek en vertalerbouw.

Dit proefschrift richt zich op hoog-niveau programmeertalen die ‘data parallel’ zijn; dat wil zeggen, de te gebruiken data in een programma kan worden benut om het onderliggende systeem aan te geven hoe het programma in parallel moet worden uitgevoerd (betekent: meerdere berekeningen tegelijkertijd doen). Het uitvoeren van een programma gebeurt in een SPMD model: Single Program Multiple Data (één programma, veelvoudige data) model. Als gevolg kan een data parallel programma parallel worden uitgevoerd, maar heeft dat niet. Daarmee bedoelen we dat zo’n programma initieel een sequentiële semantiek heeft. Dus de eerste helft van dit proefschrift wordt grotendeels besteed aan de sequentiële semantiek van een data parallelle programmeertaal.

In het algemeen zullen data parallelle programma’s parallel worden uitgevoerd. Dus gedurende het compilatie proces moeten ergens de impliciete aanwijzingen voor parallelisme omgetrokken worden naar expliciet parallelle code. Als de sequentiële semantiek gepresenteerd is gaan we dieper in op bepaalde taal constructies die een vorm van parallelisme impliceren. We beschrijven in detail de relatie tussen de semantische definities en het gebruik daarvan in de compiler.

Tenslotte presenteren we de semantiek van expliciet parallelle code; dat wil zeggen, er is niet slechts een hint voor parallelisme, maar de code schrijft expliciet een opdeling in de berekening voor.

Naast de interactie tussen semantiek en vertalerbouw zijn we geïnteresseerd in een speciaal soort compiler: een optimaliserende compiler. Zo’n optimaliserende compiler is in staat een verzameling transformaties toe te passen op een programma. Een voordeel voor deze transformaties is de beschikbaarheid van meerdere abstractieniveaus, zoals hierboven is beschreven. Op die manier kunnen transformaties gebruikt worden om om te schakelen van data parallelisme (impliciet parallelisme) naar expliciet parallelisme, en natuurlijk
ook om op ieder abstractieniveau de code te herschrijven om een efficiënter programma te verkrijgen. Met een semantisch model achter de hand bestaat de mogelijkheid de correctheid van deze transformaties te verifiëren.

De data parallelle taal die we gebruiken voor ons semantisch model en onze compiler heet *V-nus*. Het is een nieuw gedefinieerde intermediaire taal (tussentaal) die kan dienen als een platvorm waarnaar andere data parallele talen kunnen worden vertaald. Wanneer deze vertaling eenmaal is gemaakt kan gebruik worden gemaakt van de *V-nus* compiler om te experimenteren met optimalisaties door middel van de transformaties. Op dit moment is *V-nus* een volwassen taal waarvoor een semantisch model, een compiler en een verzameling transformaties beschikbaar is.
Curriculum Vitae

Paul Dechering was born on May 13, 1970, in 's-Hertogenbosch, The Netherlands. In 1982, he began with secondary school in Silvolde, named “Isala College”, and graduated in 1988. Subsequently, he started studying Computer Science at the University of Nijmegen. By 1993, he received a MSc degree and a supplementary MSc degree in Computer Science, based on theoretical examination subjects and applied examination subjects, respectively.

Thereafter, in 1993, he became a PhD student (dutch: Onderzoeker in Opleiding (OIO)) with the Dutch organization of Scientific Research (NWO), working at the Delft University of Technology.

Currently he has been employed at Hollandse Signaalapparaten BV as a Research Fellow, where he started by the end of 1997.
Index

annotation, 19, 85
APL, 5
array, see structure

back-end, 7, 15, 107
blocking (non-), 21, 101-103, 105
body, 74
   instance, 20, 65, 74
Booster, 2, 7, 22, 74, 78, 108
bounded set, see set
buffer, 99

C, 2
C++, 7
calculus, 2, 8, 107
ψ-calculus, 9
chain, 35
communication
   asynchronous, 21, 98
   channel, 97
   pattern, 9
   process, 98
   synchronous, 21, 98
compiler, 1, 6, 8, 15, 107
composition, 65-66
computes-rule, 111
concurrency, 73, 76, 97
data, see structure
declaration, 12, 18, 32, 54
decomposition, 9
dependency
   define-define, 77, 111
   define-use, 77, 111
   loop carried, 75
difference, 80
dimension, 5, 17
directive, 2, 17, 78, 90
distribution, 2, 3, 12, 20, 98, 108
domain equivalence, 69
definition, 2, 7
engine, 8, 15, 107
expression, 17
Fidel, 9
forall, 74-91
FORTRAN, 5, 7, 9, 74
front-end, 7, 15, 107
function, 29
functional language, 43
hole, 49
HPF, 5, 74

index, 4, 16
   propagation, 28, 33
   space, 74
interleaving, 97
intermediate formalism, 8, 10
intermediate language, 8-10, 15-17
ip function, see index propagation
iteration, 74
l-value, 46, 52-54, 63
lambda
   abstraction, 103, 105
   calculus, 44
least upper bound (lub), 35, 67
lexicographical order, 67
location, 17
locations, see set
match, 104

164
meaning, 1-2, 10 11, 27
    function, 43, 100, 101
memory
    free, 46
    location(s), see location(s)
NESP, 5
node, 8

optimizer, 7, 15
ordered representation, see state, ordered
    index -
orthogonality, 74
owner, 108
    computes-rule, see computes-rule
packet, 99
parallelism
    control, 3
    data, 2
    explicit, 3, 4, 20, 73, 97
    functional, 3
    implicit, 4, 20, 73
    process, 3
    task, 3
parameter
    actual, 44, 59
    formal, 44, 59
    passing, 44, 59-62
        by-name, 44
        by-reference, 59
        by-value, 44, 59
parse tree, 8
pattern, 16, 117
pragma, 85-86
procedure
    description, 44
        environment, 44
    process, 21, 97
        number, 99
r-value, 46, 53-54, 64
rank, 17
rectangular, see structure
reduce, 17, 73, 92-95
referential transparency, 19, 32, 63, 78
replacement, 28, 38, 46
rewrite rule, see transformation
scan, 17, 74, 93-95
scope, 18, 33
    dynamic, 44
    number, 60
    static, 44
semantics, 1, 10-11, 16, 27
    action, 10
    algebraic, 10
    assertions, 10
    axiomatic, 10
    compositional, 11, 43
    denotational, 10
    high-level, 16
    low-level, 16
    mathematical, 10
    operational, 10
    static, 7
set
    bounded, 30
    spanning, 41
    zero-based, 31
expression, 24
index, 9, 28
    spanning, 40
locations, 28, 30-31
    partially ordered (po), 35
        complete (cpo), 35, 67
selector, 25, 49
shift-reduce, 66
singularity term, 83
state
    communication, 99
    declaration, 32, 37 38
        index, 32, 36-37
        minimum, 67
        order on -, 45
        ordered, 45, 54, 68
        initial, 44, 46-48
location, 33
parallel program, 100
process, 98–101
program, 29, 32, 34–35
valid, see valid
view, 33–34
strategy, 107, 114
structure
   array, 4, 9, 16, 27
   flat, 5
   nested, 5
   data, 4, 17, 30
   rectangular, 4, 5, 30
syntactic application, 44
template, 16, 117
topology, 9, 108
transformation, 8, 16, 108
tree
   traversal, 114
   walk, 114
type, 17
type casting, 62
valid
   program, 26
   state, 48
variable, 17
V-cal, 2, 107
   rule, see transformation
VCODE, 5
view, 2, 9, 17, 25, 28, 30, 54–58
   function, 22, 28, 33
ZPL, 5