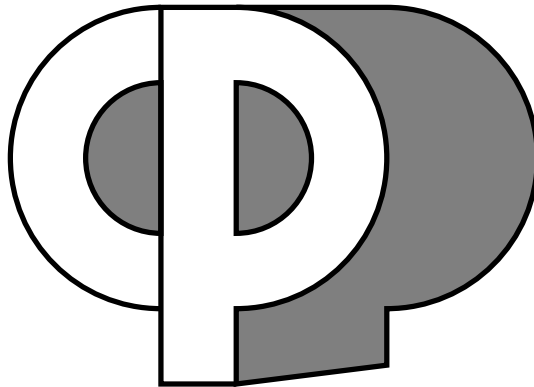


Delft University of Technology
Computational Physics Report Series

**A Generalized forall Concept
for Parallel Languages**

**Paul Dechering, Leo Breebaart, Frits Kuijman
Kees van Reeuwijk, Henk Sips**

report number CP-96-003



ISSN 1382-5224

Published and produced by:
Computational Physics Section
Faculty of Applied Physics
Delft University of Technology
Lorentzweg 1
2628 CJ Delft
The Netherlands

Information about Computational Physics Report Series:
reports@cp.tn.tudelft.nl

Information about Computational Physics Section:
<http://www.cp.tn.tudelft.nl/>

1 Introduction

The *forall* statement is an important language construct in many (data) parallel languages [2], [3], [4], [8], [11], [14]. It specifies which computations can be performed independently.

In this paper, we will define a generalized *forall* statement and discuss its semantics and implementation. We will show how *forall* constructs as found in the languages *Booster* [2], Connection Machine Fortran (CM Fortran) [4], and High Performance Fortran (HPF) [8] are mapped to this generalized *forall* statement. The *forall* statement we propose has the ability to spawn more complex independent activities than can be found in these languages. The context of our *forall* statement is supplied by *V-nus*, a concise intermediate language for data parallelism [5]. The purpose of *V-nus* is providing a language platform to which other data parallel languages can be translated, and subsequently optimized.

We will use the word *iteration* to mean any generic index-driven iteration language construct. Such a construct consists of two parts: an *index-space specification* and a *body*. 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 *body-instance*. We use the word *forall* to represent a specific kind of *iteration* that allows a concurrent execution of the body-instances. We use denotational semantics to define the meaning of the *V-nus* language constructs which will allow us to verify and optimize a *forall* statement.

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 the *forall*.
- It must be feasible 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 used in programming.

Section 2 presents an overview of different types of *iteration* statements, one of which is chosen for the *V-nus* language. In Section 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 4. In Section 5 we explain how the *V-nus forall* can be implemented efficiently. Subsequently, it is shown in Section 6 how different interpretations of a *forall* statement can be represented in *V-nus*. In Section 7 we again consider the goals of this paper. A short introduction to our compiler framework is given in Section 8. Finally, we draw some conclusions in Section 9.

2 Different types of *iteration*

In the set of *iteration* statements, we can identify two extremes: the sequential loop and the completely parallel loop.

The sequential *iteration* This *iteration* is equivalent to the conventional for-loop. The body-instances are executed one after another, in a predefined order. Data dependencies are of no consequence, since the lack of concurrency guarantees deterministic behaviour.

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 occurs in general.

There are also *iteration* statements that are neither a sequential *iteration* nor a chaos *iteration*. A number of examples of these *iteration* statements are given 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 guaranteed. 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 entire 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 use the merge *forall* in *V-nus*, because it has the most potential parallelism, and is well-suited for use in programming.

3 Existing approaches

Both data parallel languages as well as 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 on the other hand, lack of these constraints may cause non-determinism. An assignment 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 trade off are 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 analyse their effect on the program context in general.

One of the first occurrences of the *forall* statement was introduced by Thinking Machines Corporation in CM Fortran [4]. 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 cause the *forall* statement to be executed serially. The CM Fortran compiler can not currently check whether these expressions can safely be executed in parallel [4].

Vienna Fortran [14] 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 its separate copy. A body-instance can consist of any legal FORTRAN 77 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 dependence) nor written (define-define dependence) in any other body-instance (see [15] for a description of define-use and define-define dependencies). As a result of this restriction a deterministic outcome can always be computed.

The use of a *forall* statement in the Fortran dialects CM Fortran, Vienna Fortran, and Fortran D [9] 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. The HPF *forall* statement [8] only expresses an arbitrary execution order of the body-instances; it is the distribution of data that introduces parallelism.

The HPF *forall* statement consists of a single assignment statement. The left-hand side 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* body¹, but this means that each assignment of the body is executed completely; i.e. as if the assignments were written as *forall* statements in the same order (see Section 6). In addition a directive *independent* has been introduced for both *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 [7].

The data parallel language *Booster* [2] 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 [11] the *forall* statement is used to denote an array of parallel processes. A severe restriction is imposed on the *forall* body to prevent unambiguous 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++ [3], 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 non-deterministic results.

In the remainder of this paper 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.

4 The semantics

Similar to the other languages, the *V-nus forall* statement is represented by the syntax: *forall IndexSpace Body*. The term *IndexSpace* specifies the range of the index variable; the term *Body* represents the block of statements that will be executed for each value of the index variable (see Example 4.1).

¹HPF distinguishes between *forall* statements and *forall* constructs; the latter may have multiple statements in their bodies.

Example 4.1 *The V-nus forall statement*

Consider: forall [i:3] {a := i}. The index variable is i and ranges over 0, 1 and 2. The body is a := i; an example of a body-instance is a := 1.

Body-instances of the *V-nus forall* statement are to be executed completely independently. 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 dependence between the possible body-instances; i.e. two body-instances that write to the same variable. We say that

a *forall* statement is deterministic if no define-define dependence is present between any two different body-instances of the *forall* statement.

We want to record the concept of the *forall* statement in a semantic model, such that we can use this model to reason about a program. We use denotational semantics [1] [13], in which the meaning of a program can be expressed by the composition of the meanings of its parts. The denotational semantics are useful when we want to rewrite only parts of a program, and leave the meaning of the whole program as it is.

In denotational semantics a program state captures all necessary information about the context in which a program fragment is executed. A program state is valid only if each variable of the program state is given exactly one value (see Example 4.2).

Example 4.2 *Program states.*

Consider the *forall* statement of Example 4.1. A valid program state after execution of the body-instance $a := 1$ is: $(i = 1, a = 1)$. The program state $(i = 1, a \in \{0, 1\})$ is invalid, because the variable a is given two values.

The semantics of a program fragment are given by a program state change, represented by a pair (ps, ps') 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 (ps, ps_i) is computed. Then the different program states ps_i (for all i) are merged into the final program state ps' , which will be the program state after the *forall* statement has been executed. This merge operation consists of two actions. First ps_i is compared with ps , providing only the difference $diff_i$ between these program states. Secondly, all elements of $diff_i$ will be put into ps . This is done for all ps_i in arbitrary order.

5 The implementation

Implementing the *forall* statement as presented in Section 4 may cause some 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.

In order to arrive at an efficient implementation of the *forall* statement, we take the following approach. At the start of a *forall* statement the program state ps is preserved. For the execution of a body-instance a subset ps_i of ps is used for the context in which this body-instance will be executed. Only the data that is needed in the body-instance is extracted from ps and will be used for ps_i . Each time something needs to be read from memory, it is read from ps_i . When something needs to be written to memory, it is not only stored in ps_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 1).

In the implementation of a deterministic *forall* statement, all differences between the program states ps_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* statements. That makes the result of a non-deterministic *forall* statement dependent on the computation order. In this case the efficient implementation of a *forall* statement may compute other results than the theory prescribes. In Example 5.1 a possible difference is presented between the computation used in the implementation, and the computation used in the semantics.

Example 5.1 *Difference between theory and implementation.*

Consider the program fragment: `forall [i:2] {a := i; b := i}`. The denotational semantics predict that the body-instance for $i = 0$ will result in the program state $ps_0 = (a = 0, b = 0)$. The body-instance for $i = 1$ will result in the program state $ps_1 = (a = 1, b = 1)$. ps' will then be either ps_0 or ps_1 .

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

- $a := 0, b := 0, a := 1, b := 1$ or $a := 1, b := 1, a := 0, b := 0$ or
- $a := 0, a := 1, b := 0, b := 1$ or $a := 1, a := 0, b := 1, b := 0$ or
- $a := 0, a := 1, b := 1, b := 0$ or $a := 1, a := 0, b := 0, b := 1$

which will lead to the same possible program states as predicted by the theory, *plus* the program states $(a = 0, b = 1)$ and $(a = 1, b = 0)$.

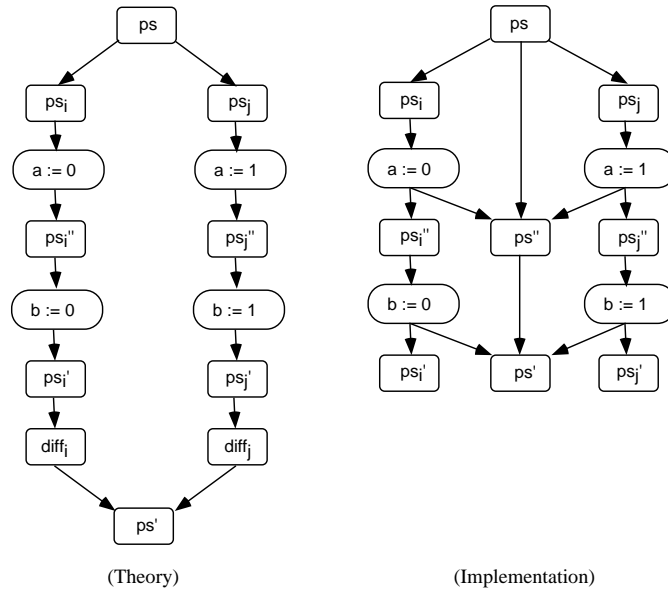


Figure 1: Program state changes caused by a *forall* statement

In Example 5.1 both the body-instances write to the variables a and b , which makes the *forall* statement non-deterministic. Theory and implementation only differ for non-deterministic *forall* statements.

6 The *forall* compared

As shown in Section 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* statements of the data parallel languages *Booster*, CM Fortran, and HPF, and we show how these differently defined *forall* statements can be mapped to the *V-nus forall* statement.

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.

Consider the following examples in pseudo code:

```
forall i=0,n j=0,m
  a[i,j] = expr
end
```

(6.1)

```
forall i=0,n j=0,m
  a[i,j] = F(X)
end
```

(6.2)

```
forall i=0,n j=0,m
  a[i,j] = expr,
  a[i+1,j] = F(X)
end
```

(6.3)

where the expressions n and m are not dependent on each other, *expr* is some arbitrary expression that does not contain a function call, F represents a function, and X is an actual argument list that is not dependent on the array a . In each of the languages *Booster*, CM Fortran, and HPF the index space over which is iterated is the Cartesian product $[0 \dots n] \times [0 \dots m]$.

In CM Fortran, Example 6.1 will cause the assignments to be executed on the CM in parallel. The assignments of Example 6.2 will be executed sequentially because of the function call on the right hand side. Example 6.3 is not valid since CM Fortran allows only one statement in a *forall* body.

In *Booster*, both Example 6.1 and Example 6.2 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 6.3 invalid.

In HPF, Example 6.1 and 6.2 have the same meaning as in *Booster*. Although functions in HPF need not to be referentially transparent, it is forbidden for functions to have side effects. This allows the different body instances of a *forall* statement to be evaluated in arbitrary order. Example 6.3 is semantically equivalent to the following consecutive *forall* statements:

```
forall i=0,n j=0,m a[i,j] = expr.
forall i=0,n j=0,m a[i+1,j] = F(X)
```

Note that the second *forall* statement only starts when the first *forall* statement has finished. It can not be rewritten to one do independent loop, because a define-define dependence exists for $a[i]$, $1 \leq i \leq n - 1$.

Example 6.1 interpreted in *Booster*, CM Fortran, or HPF can be represented in *V-nus* by:

```
forall [i:n+1, j:m+1] {a[i,j] := expr}
```

Example 6.2 interpreted in CM Fortran needs a sequential loop in *V-nus*, such as:

```
for [i:n+1, j:m+1] {a[i,j] := F(X)}
```

In *Booster* and HPF this example can be represented in the same way as Example 6.1 is represented. Example 6.3 interpreted in HPF can be rewritten to two single assignment *forall* statements as presented above. These can easily be translated to *V-nus*. Note that if Example 6.3 was interpreted in *V-nus* directly, it would denote a non-deterministic *forall* statement 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 6.3 is replaced by i then the *forall* statement has become

deterministic. Note that every do independent loop in HPF can be represented by the *V-nus forall* statement, since no loop carried dependencies occur at all.

Now, we show an example of an optimization that can only be expressed in *V-nus*. Consider the following matrix operation:

```
for [j:m] {
  forall [i:n] {a[i,j] := a[i,j-1] + a[i,j+1] + a[i-1,j] + a[i+1,j] }
}
```

The optimization we have in mind is based on synchronization elimination [10]. By reversing the *i* and *j* loop the operation can be expressed as

```
forall [i:n] {
  for [j:m] {a[i,j] := a[i,j-1] + a[i,j+1] + a[i-1,j] + a[i+1,j] }
}
```

which has no computational differences in the result. Instead of executing *forall* statements 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 dependence occurs, which makes it a deterministic *forall* statement. This *forall* statement is not ‘valid’ in the other parallel languages mentioned in this paper.

7 The goals revisited

We can now show that we have met the four requirements for the generalized *forall* statement, as listed in Section 1.

denotational semantics For non-deterministic *forall* statements an unambiguous program state change is forced by the specification of a computation order; i.e. the order in which the program states ps_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 $diff_i$ and $diff_j$ contain the same variable x . That means that both in ps_i and in ps_j the variable x has been changed with respect to the original ps . That can only be so, if body-instance i has a ‘define’ for x as well as body-instance j . But for deterministic *forall* statements this can never occur. So, there are no two different $diff_i$ and $diff_j$ containing the same variable. When all the $diff_i$ are put into ps for the construction of ps' , ps' will be a valid program state. Note that it does not matter in which order the $diff_i$ are inserted into ps – the same ps' 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.
- During execution of a body-instance, each write action must be performed twice in order to be visible in the program state of this body-instance as well as in the global program state (in some cases, the two write actions can be reduced to one).

The computation time for the construction of the program state ps' 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.

representation *V-nus* can be used to capture the meaning of different definitions of *forall* statements, as is shown in Section 6. Therefore, we think that our *forall* definition is suitable for an intermediate representation.

description 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 in 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 dependence, when different values are written to the same variable. When the same value is written twice to that variable a define-define dependence 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 dependence that causes non-determinism.

8 The *V-nus* compiler

Traditional compilers for data parallel languages advocate the ‘one tool does all’ approach: parsing, optimizing, and code generation are strongly interweaved and they are often hard-coded in the compiler. In the compiler that we have constructed, we have taken a different approach where all these phases are handled by separate tools (see Figure 2).

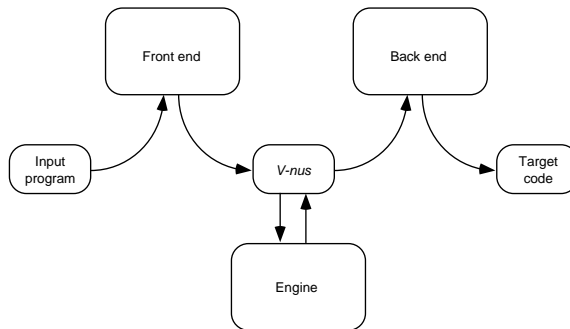


Figure 2: Overview of the *V-nus* compiler

Our experimental compiler consists of three tools [6] [12]: a front end that parses the input program, an optimizer that operates on the intermediate language *V-nus*, and a back end that generates target code. At this moment we have a front end that compiles *Booster* to *V-nus*. The back end can produce target code for shared memory systems and distributed memory systems, both being able to process the *V-nus forall* statement as it is presented in this paper.

9 Conclusion and future work

We have constructed a general *forall* statement in the intermediate language *V-nus*, capable of representing *forall* statements of other data parallel languages. The *forall* statement has a semantics that is easy to understand and is unambiguous. The compiler or execution tracing tools can check whether the requirements are met that make the *forall* statement deterministic. The body of this *forall* statement has the same syntax as an ordinary loop body. This allows the spawning of more complex concurrent computations than can be found in other data parallel languages. Furthermore, we have shown it is still possible to have an implementation without a major increase

of computation overhead. In this way, our *forall* statement is suited to be used as an intermediate representation and as a language construct in data parallel languages.

We have shown that *V-nus* can be used as an intermediate representation for different *forall* statements by using three examples. In the near future we will investigate a generalized translation scheme for *forall* statements in the languages *Booster*, CM Fortran, and HPF to *V-nus*. In an unpublished short note we have compared more *forall* examples than presented in this paper. This note is available at:

`ftp://ftp.cp.tn.tudelft.nl/pub/cp/publications/1996/ForallExamples.ps.Z`

References

- [1] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Series in Computer Science. Prentice Hall International, 1980.
- [2] L.C. Breebaart, P.F.G. Dechering, A.B. Poelman, J.A. Trescher, J.P.M. de Vreught, and H.J. Sips. The Booster Language, Syntax and Static Semantics. Computational Physics report series CP-95-02, Delft University of Technology, 1995.
- [3] P. Carlin, M. Chandy, and C. Kesselman. The Compositional C++ Language Definition. Revision 0.9 `ftp://ftp.compbio.caltech.edu/pub/CC++/Docs/cc++-def`, March 1 1993.
- [4] Thinking Machines Corporation. CM Fortran Programming Guide. Technical report, January 1991.
- [5] P.F.G. Dechering. The Denotational Semantics of Booster, A Working Paper 2.0. Computational Physics report series CP-95-05, Delft University of Technology, 1995.
- [6] P.F.G. Dechering, J.A. Trescher, and J.P.M. de Vreught. V-cal: a Calculus for the Compilation of Data Parallel Languages. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 111–125, Columbus, Ohio, USA, 1995. Springer Verlag.
- [7] Fortran Forum. Special Issue, Fortran95, Committee Draft, May 95. *Fortran Forum*, 12(2), 1995.
- [8] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical report, November 1994.
- [9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. COMP TR90079, Department of Computer Science, Rice University, March 1991.
- [10] A.J.C. Gemund. *Performance Modelling of Parallel Systems*. PhD thesis, Delft University of Technology, 1996.
- [11] P.B. Hansen. Interference Control in SuperPascal – A Block-Structured Parallel Language. *The Computer Journal*, 37(5):399–406, 1994.
- [12] J.A. Trescher, L.C. Breebaart, P.F.G. Dechering, A.B. Poelman, J.P.M. de Vreught, and H.J. Sips. A Formal Approach to the Compilation of Data Parallel Languages. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 155–169, Ithaca, New York, USA, 1994. Springer Verlag.
- [13] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, 1993.

- [14] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification, version 1.1. Internal Report 21, ICASE, 1992.
- [15] H. Zima and B. Chapman. *Supercomputers for Parallel and Vector Computers*. Frontier Series. Addison-Wesley, 1990.