A Generalized **forall** Concept for Parallel Languages

P.F.G. Dechering, L.C. Breebaart, F. Kuijlman,
C. van Reeuwijk, H.J. Sips

BoosterTeam@cp.tn.tudelft.nl
Delft University of Technology
The Netherlands

Extended abstract

The *forall* statement is an important language construct in many (data) parallel languages [1], [2], [3], [6], [8], [9]. It gives an indication to the compiler which computations can be performed independently.

In this abstract, we will define a generalized *forall* statement and discuss its implementation. This *forall* statement has the ability to spawn more complex independent activities than can be expressed in these languages. Existing *forall* statements can be mapped to this generalized concept. The context of our *forall* statement is supplied by *V-nus*, a concise intermediate language for data parallelism [4]. The purpose of *V-nus* is providing a language platform to which other data parallel languages can be translated, and subsequently optimized.

Our *forall* statement consists of two parts: an *index-space specification* specifying the range of the index variable, and a *body* representing a block of statements. 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 denotational semantics to define the meaning of the *V-nus* language constructs. With these we can verify and optimize a *forall* statement.

It has been our goal to find a *forall* statement that complies with the following requirements: (1) 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*. (2) 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. (3) The *forall* statement must be capable of representing a wide class of *forall* definitions as can be found in (data) parallel languages. (4) It must be possible to give a concise operational semantics of the *forall* statement that can easily be used in programming.

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 should not affect the outcome 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 use denotational semantics, in which the meaning of a program can be expressed by the composition of the meanings of its parts, to record the concept of the *forall* statement. The semantics are described by using a difference and a merge operation on program states [5]. 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. Every 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 be used by 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.

The construction of ps_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 dependence 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* statement specify which data should be copied in ps_i.

If a *forall* statement is not annotated by a pragma, then the local program states ps_i are created as explained above. If a pragma is present the compiler relies on this information and only copies the given data structures for the accompanying program states ps_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* statement is:

<< dependsOn *Expression* >>

which expresses a dependency for the data structure(s) *Expression*. An empty list of specifications (i.e. << >>) 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.

We end this abstract with an example of an optimization that can only be expressed by using the *V-nus forall*. 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 [7]. 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 we refered to in this abstract.

More detailed information regarding the generalized *forall* concept can be found in our technical report [5] available at:

ftp://ftp.cp.tn.tudelft.nl/pub/cp/publications/1996/CP-96-003.ps.Z

References

1. 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.
2. 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.
3. Thinking Machines Corporation. CM Fortran Programming Guide. Technical report, January 1991.
4. 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.
5. P.F.G Dechering, L.C. Breebaart, F. Kuijlman, C. van Reeuwijk, and H.J. Sips. A Generalized forall Concept for Parallel Languages. Computational Physics report series CP–96–003, Delft University of Technology, 1996.
6. High Performance Fortran Forum. High Performance Fortran Language Specification. Technical report, November 1994.
7. A.J.C. van Gemund. *Performance Modelling of Parallel Systems.* PhD thesis, Delft University of Technology, 1996.
8. P.B. Hansen. Interference Control in SuperPascal – A Block-Structured Parallel Language. *The Computer Journal*, 37(5):399–406, 1994.
9. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification, version 1.1. Internal Report 21, ICASE, 1992.

This article was processed using the LaTeX macro package with LLNCS style