In *Proceedings of the Joint Symposium on Parallel Processing 1993*, pp. 269–276, Waseda University, Tokyo, Japan, May 17–19, 1993.

MODULA-3*: AN EFFICIENTLY COMPILABLE EXTENSION OF MODULA-3 FOR PROBLEM-ORIENTED EXPLICITLY PARALLEL PROGRAMMING

Ernst A. Heinz, University of Karlsruhe, F.R. Germany
(email: heinze@ira.uka.de)

Abstract

In this paper we present the programming language Modula-3 and machine-independent optimization techniques for its compilation. We consider Modula-3* to be a promising new framework for problem-oriented explicitly parallel programming that aims at high performance on a variety of architectures and machines.*

By proposing a large collection of effective optimizations we illustrate the feasibility of efficient Modula-3 compilation. We ensure portability of the compiler by formulating most optimizations as source-to-source transformations in the intermediate high-level language Modula-3π, an extension of Modula-3* especially designed for this task.*

1 Introduction

Problem-oriented explicitly parallel programming environments (languages, compilers, debuggers) will lead the way to widespread acceptance and successful use of highly parallel computers. The most pressing concerns of this field are portability and performance.

Portability is ensured by high-level language constructs and standard library interfaces that guarantee machine-independence. The needs of high performance call for sophisticated optimizing compilers fine-tuned for each target machine. Good performance of compiled codes must be verified on various parallel architectures and machines in order to demonstrate the practical usefulness of portable parallel programs.

Taking the above considerations into account we propose Modula-3* [6] as a promising new framework for problem-oriented explicitly parallel programming. With regard to parallel language constructs Modula-3* is an evolution of Modula-2* [12] which was defined by Tichy and Herter in 1990. Modula-3* offers far greater flexibility than its predecessor by supporting mixed-mode parallel computations, local declarations of processes, and massively parallel object-oriented programming. We illustrate the expressiveness of Modula-3* and discuss in detail how to keep optimizing Modula-3* compilers portable.

The paper proceeds with an introduction of Modula-3* in Section 2. Our proposal of portable optimizing Modula-3* compilation follows in Section 3. Thereafter, we describe a machine-independent intermediate language designed for optimizing Modula-3* compilation in Section 4. Finally, we present selected optimizations in Section 5 and conclude with some prospective comments on future work.

2 Modula-3*

Modula-3* is an extension of Modula-3 [2] for problem-oriented explicitly parallel programming. The programming model combines control and data parallelism. The compact new features are easy to learn, easy to use, and equally well suited for other imperative programming languages, such as Fortran, C, or Ada.

Modula-3* encourages extensive use of parallel constructs by supporting all Modula-3 abstraction mechanisms for parallel programming, too. Arbitrary nestings of synchronous, asynchronous, and mixed-mode parallel computations operate on data in the same single address space. Procedures may be called in any context, whether sequential or parallel, at any nesting depth and allow for recursive parallelism without restriction.

2.1 FORALL Statement

In Modula-3* you invoke parallel activity by means of the `FORALL` statement.

```
FORALL Id ":" Type DO
  S
END
```

A `FORALL` creates as many (conceptual) processes as there are elements in the possibly non-static scalar range *Type*. The identifier *Id* is local to the

FORALL and serves as a runtime constant; it is initialized to a unique value of *Type* for each process.

Modula-3* FORALL statements are synchronous by default. The created processes execute statement sequence S in MSIMD-like parallel synchrony. Except for procedure calls, the semantics of synchronous execution are defined as in Modula-2* [12]. In case of synchronous procedure calls Modula-3* only requires the synchronization of parameter passing. The statements inside called procedures need not be synchronized.

Furthermore, the END of a FORALL statement imposes an explicit synchronization barrier on the created processes. Termination of the whole FORALL is delayed until all processes have finished their execution of S.

The following example illustrates syntax and semantics of FORALL statements. It implements a parallel permutation of vector X according to permutation function p().

```
FORALL i: [1..n] DO
  X[i] := X[p(i)];
END
```

The synchronous semantics ensure that all rhs elements X[p(i)] are read and temporarily stored before any lhs variable X[i] is written. Thus, parallel modifications of overlapping data structures are easily formulated in Modula-3*.

To increase readability and reduce source code length Modula-3* supports a compact shorthand notation for directly nested FORALLs. The grammar rules of Appendix A.1 define its exact syntax. A parallel matrix transpose using this notation is shown below.

```
FORALL i, j: [1..n] DO
  A[i,j] := A[j,i];
END
```

2.2 NOSYNC Statement

In addition to synchronous parallelism Modula-3* supports asynchronous and mixed-mode parallel programming by providing the NOSYNC statement.

```
NOSYNC
  S
END
```

A NOSYNC has no semantic meaning in sequential and asynchronous contexts. In synchronous contexts, however, NOSYNCs disable synchronization for the entering processes during their execution of statement sequence S. Synchronization is re-established by an explicit synchronization barrier at the END. Termination of the whole NOSYNC is delayed until all processes of the entering set have finished their execution of S.

For the sake of readability and compactness Modula-3* supports a convenient shorthand notation for completely asynchronous FORALLs. The grammar rules of Appendix A.1 define its exact syntax. Using this notation the vector addition $Z = X + Y$ can be formulated in parallel for non-overlapping arrays X, Y, and Z as follows.

```
FORALL i: [1..n] NOSYNC DO
  Z[i] := X[i] + Y[i];
END
```

2.3 Parallel Libraries

The Modula-3* parallel libraries [6] aim at scalability, portability, and efficiency of frequently used parallel operations like enumeration, reduction, scan, sort, parallel I/O etc. Scalability means that the library routines operate on open array parameters of arbitrary size. Portability is ensured by providing the same Modula-3* interfaces on all target machines. To achieve efficiency special low-level features of each target machine are exploited in the different library implementations.

Another interesting feature of the Modula-3* parallel libraries is their functional diversity. Wherever possible, normal, masked, segmented, and universal (masked plus segmented) versions of the parallel operations are provided.

2.4 Programming Examples

If placed inside FORALLs, Modula-3* block statements introduce declarations that are local to each process. The following program fragment implements an element-wise swap of the two vectors X and Y. The local variable Xi of process i serves as a temporary to store the initial value of X[i].

```
FORALL i: [1..n] NOSYNC DO
  VAR Xi := X[i];
  BEGIN
    X[i] := Y[i];
    Y[i] := Xi;
  END;
END
```

Local declarations simplify parallel programming and ease the analysis of parallel code. Furthermore, they influence performance. Especially local variables of processes are often crucial for the efficiency of parallel algorithms because of their collocation with computation.

As another example we consider standard methods for numerical solution of partial differential equations that frequently employ Red-Black successive over-relaxation (SOR). The kernel of each Red-Black iteration comprises two successive asynchronous steps. A parallel Modula-3* implementation with `f()` denoting the relaxation function is presented below.

```
FORALL i, j: [1..n] DO
  NOSYNC                      (* red points *)
    IF (i MOD 2)=(j MOD 2) THEN
      A[i,j] := f(A[i-1,j], A[i,j-1],
                  A[i+1,j], A[i,j+1]);
    END;
  END;
  NOSYNC                    (* black points *)
    IF (i MOD 2)#(j MOD 2) THEN
      A[i,j] := f(A[i-1,j], A[i,j-1],
                  A[i+1,j], A[i,j+1]);
    END;
  END;
END
```

Last but not least we present an image processing problem and its parallel solution in Modula-3*. The tasks are to smooth a symmetric 2D image by application of a spatial filter, to store the smoothed image for later use, to distort the smoothed image by combination with a distortion matrix, to mirror the distorted image at its diagonal, to additively combine the mirrored image with the un-smoothed original yielding a new image, and, finally, to transpose the distortion matrix.

Let A denote the pixel matrix of the original image, B the original distortion matrix, C the pixel matrix of the resulting image, and f the function for spatial filtering. Then, the above tasks can be solved by calculating A', B', and C as follows.

$$
\begin{aligned}
A' &= f(A) \\
B' &= B^T \\
C &= A + \big(f(A) * B\big)^T
\end{aligned}
$$

In Modula-3* we use two-dimensional `REAL` arrays for the original image `A`, the distortion matrix `B`, and the new image `C`. The spatial filter is implemented by a function procedure `f()`. The proposed solution incorporates mixed-mode parallel programming and local variables of processes.

```
FORALL i, j: [1..n] DO
  VAR Aij := A[i,j];       (* save original *)
      Pij := 0.0;
  BEGIN                    (* smooth original *)
    A[i,j] := f(A[i-1,j], A[i,j-1],
                A[i+1,j], A[i,j+1], A[i,j]);
    NOSYNC
      FOR k := 1 TO n DO   (* calc. product *)
        INC(Pij, A[i,k]*B[k,j]);
      END;
      C[j,i] := Pij;   (* mirror by transp. *)
    END;
    C[i,j] := Aij + C[i,j];    (* add orig. *)
    B[i,j] := B[j,i];    (* transp. distor. *)
  END;
END
```

3 Compilation

The results of [10, 11, 13] suggest the feasibility of efficient Modula-3* compilation for different target machines. Because performance of compiled codes must be verified on a variety of architectures and machines the compiler itself should be as portable as possible. Instead of generating native code, we agree with [10] and recommend to generate C augmented with machine-dependent macros.

The experiences of problem-oriented compilation for parallel machines [8, 9, 13] show that many important optimizations rely on high-level restructuring techniques. To name but a few, these optimizations include automatic data and computation alignment, automatic data distribution, and automatic synchronization barrier elimination (see Section 5 for details). The desired restructurings simply result in source-to-source transformations if the underlying languages are intentionally designed with this option in mind. Because the compact features of Modula-3* itself cannot fulfill this demand we developed Modula-3π [5] to serve as a high-level intermediate language for portable optimizing Modula-3* compilation. Modula-3π is introduced in Section 4. Basically, it augments Modula-3* with machine-independent allocation and alignment specifications.

To achieve good performance on different architectures and machines we propose the following optimization phases for Modula-3* compilation. After initial Modula-3*–to–Modula-3π translation most phases may be performed by Modula-3π–to–Modula-3π transformations. Section 5 describes some of the optimizations in more detail.

FORALL Combination. Equal range and directly nested `FORALL`s are transformed to shortest possible form.

Synchronization Elimination. After elimination of redundant synchronization barriers synchronous `FORALL`s are transformed to equivalent asynchronous `FORALL`s. After the transformation each `FORALL` is in "loosely synchronous form" containing a sequence of `NOSYNC` statements at the top nesting level.

Data Alignment. Arrays accessed in parallel are aligned with one another based on a global analysis of index references.

Computation Alignment. Parallel computations are aligned with data based on an analysis of index expressions inside `FORALL`s.

Data Allocation. All dimensions of each array are decorated with allocators based on a global analysis of index references.

Data Redistribution. Between blocks of parallel computations data may be redistributed based on a global analysis of index references.

Access Vectorization. Data references causing remote access in parallel computations are accumulated and vectorized based on liveliness analysis inside `FORALL`s.

Synchronization Diversification. Data redistribution and remote data access may be incorporated into synchronization barriers based on a global data dependence analysis.

Latency Hiding. Remote access latency caused by network and communication delays may be hidden by overlapping local computation and non-local data access.

FORALL Balancing. Bodies of `FORALL` statements may be replicated, shrunk, or even omitted based on branch, loop, and procedure call prediction in parallel computations.

Fast Communication. Special communication mechanisms of the target machines should be exploited to increase remote access efficiency.

Process Virtualization. Parallel computations are automatically virtualized by simulating the necessary number of processes on the physically available processors.

Process Balancing. Dynamic and static processor loads may be balanced based on the predictions of FORALL balancing.

4 Modula-3π

Modula-3π extends Modula-3* by dynamic computation alignment, static data alignment, and static data allocation. All new features guarantee machine-independence because of their problem-oriented formulation.

The basic idea of alignment is to describe problem spaces and the placement of data as well as computation therein with respect to one another. Computation and data alignment reduce communication overhead while preserving the problem-inherent parallelism.

Machine-independent data allocation specifies the "affinity" of adjacent elements and dimensions of arrays. Thus, it guides the distribution of data over the actually available processors.

4.1 ALIGNED WITH Statement

In Modula-3π you align computation with data by means of the `ALIGNED WITH` statement.

```
ALIGNED WITH Expr DO
  S
END
```

The alignment expression *Expr* of an `ALIGNED WITH` specifies the data that must be accessible at minimal cost during execution of statement sequence S. No semantic meaning is implied.

A sequential `ALIGNED WITH` tells the compiler how to align sequential execution. In parallel contexts, however, `ALIGNED WITH`s refer to each participating process separately. Different alignment of all processes results if the alignment expressions are suitably indexed.

```
FORALL i: [1..n] NOSYNC DO
  ALIGNED WITH Z[i] DO
    Z[i] := X[i] + Y[i];
  END;
END
```

The above is an extension of the vector addition example from Section 2.2. For each process i the `ALIGNED WITH` declares vector element `Z[`i`]` to be the data to align with during component addition.

4.2 Data Alignment

Modula-3π supports static data alignment by providing the `ALIGN` declaration.

```
[FORALL IdDecl {IdDecl}]
  ALIGN Expr {"," Expr} WITH Expr
```

The expression list between the `ALIGN` and `WITH` keywords specifies the data that is to be aligned. The single expression at the end represents the data to align with. The optional `FORALL` allows the declaration of index range identifiers whose scope are the expressions of the `ALIGN` clause. Non-contiguous and offset alignments need this sort of "quantification".

In Modula-3π data may only be aligned once. The corresponding `ALIGN` declaration must be included in the same static scope as the data declaration itself. Realignment, alignment cycles, and partial alignment are not allowed. Furthermore, Modula-3π requires all expressions of `ALIGN` declarations to be evaluatable at compile time.

The simplest form of data alignment occurs if exact adjacency is required. This is certainly true for the arrays `X`, `Y`, and `Z` of the vector addition example.

```
ALIGN X, Y WITH Z;
```

In combination with the aligned vector addition of Section 4.1 this `ALIGN` declaration results in perfect data and computation alignment.

An example of more sophisticated data alignment requiring quantified `ALIGN` declarations follows below. There, the quantified `ALIGN` specifies offsetted alignment of vector `X` and non-contiguous alignment of vector `Y` with respect to vector `Z`.

```
FORALL i: [1..n]
  ALIGN X[i-2], Y[4*i] WITH Z[i];
```

4.3 Type Alignment

Beside data and process alignment Modula-3π also allows for static type alignment by accepting type expressions in `ALIGN` declarations. Types are a natural representation of problem spaces. Hence, type alignment is actually equivalent to the alignment of problem spaces.

Moreover, type alignment permits data placement within problem spaces. To do so you simply align variables with types. All in all Modula-3π covers data alignment, problem space alignment, and data placement by one single construct: the `ALIGN` declaration.

The following example illustrates type alignment, data placement, and related syntactic issues.

```
TYPE T = ARRAY[1..2*n], [1..2*n] OF REAL;
     U = ARRAY[1..n], [1..n] OF REAL;

FORALL i, j: [1..n]
  ALIGN U[i,j] WITH T[n+i,n+j];

VAR  A : ARRAY[1..n], [1..n] OF REAL;

FORALL i, j: [1..n]
  ALIGN A[i,j] WITH T[2*i,2*j];
```

4.4 Data Allocation

In Modula-3π machine-independent data allocation relies on so-called "allocators" which are assigned to dimensions of arrays.

```
ARRAY Type [Alloc] {"," Type [Alloc]} OF Type

ARRAY "(" Type {"," Type} ")" Alloc OF Type

Alloc = LOCAL | SPREAD | CYCLIC ["(" Expr ")"]
```

The names and meanings of the three different allocators `LOCAL`, `SPREAD`, and `CYCLIC` resemble those of their Modula-2* [12] and Fortran D [3] counterparts. If no allocator is given it defaults to `LOCAL` for the corresponding dimension.

LOCAL: The meaning of `LOCAL` allocation is that array elements are spatially collocated during data distribution if their indices differ in `LOCAL` dimensions only. Therefore, dimensions with allocator `LOCAL` are normally assigned to single processors.

SPREAD: When using `SPREAD` allocation index adjacency of array elements leads to spatial collocation during data distribution. Dimensions with allocator `SPREAD` are distributed in contiguous chunks.

CYCLIC: With `CYCLIC` allocation index adjacency of array elements leads to spatial dislocation during data distribution. Dimensions with allocator `CYCLIC` are distributed slicewise in a round-robin fashion. The optional *Expr* specifies the size of the slices. It must be a compile-time constant.

Modula-3π supports two allocation variants that differ in syntax and semantics. The first variant allocates each dimension separately and requires exactly one allocator per dimension.

```
VAR A: ARRAY [1..n] SPREAD,
             [1..n] SPREAD OF REAL;
```

The second variant handles combinations of dimensions and requires one allocator per combination.

```
VAR B: ARRAY ([1..n],
              [1..n]) SPREAD OF REAL;
```

As for allocators, Modula-3π treats open arrays like normal arrays. Only the syntax of dimension combination differs slightly.

```
VAR AA: ARRAY SPREAD OF ARRAY SPREAD OF REAL;

VAR BB: ARRAY (OF ARRAY) SPREAD OF REAL;
```

Combined dimensions are conceptually unrolled row-wise before allocator application. The above matrices `A` and `B` would therefore be distributed over four processors $p_1, \ldots, p_4$ as follows.

p_1: A$[1 : n/2, 1 : n/2]$ B$[1 : n/4, 1 : n]$

p_2: A$[1 : n/2, n/2 + 1 : n]$ B$[n/4 + 1 : n/2, 1 : n]$

p_3: A$[n/2 + 1 : n, 1 : n/2]$ B$[n/2 + 1 : 3n/4, 1 : n]$

p_4: A$[n/2 + 1 : n, n/2 + 1 : n]$ B$[3n/4 + 1 : n, 1 : n]$

4.5 Example

In general, Modula-3π programs contain more information about arrays and parallel computations than equivalent Modula-3* programs. The following result-equivalent Modula-3* and Modula-3π program fragments illustrate the information surplus of Modula-3π.

```
(*-------- Modula-3* --------*)

VAR X, Y: ARRAY [0..n+1] OF REAL;

BEGIN
  FORALL i: [1..n] DO
    X[i-1] := X[i] + Y[i+1];
  END;
END
```

$\updownarrow$

```
(*-------- Modula-3π --------*)

VAR X, Y: ARRAY[0..n+1] SPREAD OF REAL;

FORALL i: [1..n]
  ALIGN Y[i+1] WITH X[i-1];

BEGIN
  FORALL i: [1..n] DO
    ALIGNED WITH X[i-1] DO
      X[i-1] := X[i] + Y[i+1];
    END;
  END;
END
```

5 Selected Optimization Techniques

5.1 FORALL Combination

The intentions and potential benefits of FORALL combination are quite obvious. A typical example follows below.

```
FORALL i: [1..n] DO
  FORALL j: [1..n] DO
    A[i,j] := A[i,i] + B[j,j];
  END;
END;
FORALL i: [1..n] DO
  FORALL j: [1..n] DO
    B[i,j] := B[j,i];
  END;
END
```

$\downarrow$

```
FORALL i, j: [1..n] DO
  A[i,j] := A[i,i] + B[j,j];
  B[i,j] := B[j,i];
END
```

5.2 Synchronization Elimination

The conservative synchronous semantics defined in [6] prescribe a vast number of implicit synchronization barriers for synchronous **FORALL** statements. Most real synchronous **FORALL**s, however, only need a fraction thereof to ensure correctness.

According to [7] automatic elimination of redundant synchronization barriers is possible. For detection you apply standard data dependence analysis [1, 4] to **FORALL**s. The elimination reduces the number of synchronization barriers by more than 50% on average. This obviously pays off for machines without hardware-supported barrier synchronization. All other machines, including SIMD, benefit from synchronization barrier elimination, too, because it reduces the overhead of process virtualization and may improve register usage.

Below we demonstrate the effectiveness of synchronization barrier elimination by comparing two different transformation results of a simple synchronous **FORALL** statement. On the lower left you see an un-optimized loosely synchronous version of the **FORALL** that makes all implicit synchronization barriers explicitly visible. The elimination algorithm determines three of five synchronization barriers as redundant. The resulting optimized loosely synchronous version of the **FORALL** is shown on the lower right.

```
FORALL i: [1..n] DO
  X[i] := X[p(i)];
  Y[i] := Y[i+1] + X[i];
END
```

$\swarrow$ $\searrow$

```
FORALL i:[1..n] DO
  VAR H:REAL;
  BEGIN
    NOSYNC
      H:=X[p(i)];
    END;
    NOSYNC
      X[i]:=H;
    END;
    NOSYNC
      H:=Y[i+1];
    END;
    NOSYNC
      H:=H+X[i];
    END;
    NOSYNC
      Y[i]:=H;
    END;
  END;
END
```

```
FORALL i:[1..n] DO
  VAR H1,H2:REAL;
  BEGIN
    NOSYNC
      H1:=X[p(i)];
      H2:=Y[i+1];
    END;
    NOSYNC
      X[i]:=H1;
      Y[i]:=H2+X[i];
    END;
  END;
END
```

5.3 Access Vectorization

Access vectorization, synchronization diversification, and latency hiding all aim at the reduction of communication overhead caused by remote data access. These optimization techniques are nonetheless completely independent of one another. The best optimization results, however, are always achieved by an intelligent combination of all three.

Due to space limitations we can only show one of the three optimizations in action. We choose to present access vectorization. The following example illustrates how this optimization works.

```
FORALL i: [1..n] NOSYNC DO
  Z[i] := X[i] + Y[2*i];
  A[i,i] := Y[2*i+1] + Z[i];
  R[i,i,i] := Y[i] + Y[2*i-1];
END

        ↓

ALIGN X, Y WITH Z;

BEGIN
  FORALL i: [1..n] NOSYNC DO
    ALIGNED WITH Z[i] DO
      VAR H1, H2, H3, H4, H5: REAL;
      BEGIN
        (* vectorized remote reads *)
        BEGIN
          H1 := Y[2*i-1];
          H2 := Y[2*i];
          H3 := Y[2*i+1];
        END;
        (* local computations *)
        Z[i] := X[i] + H2;
        H4 := H3 + Z[i];
        H5 := Y[i] + H1;
        (* vectorized remote writes *)
        BEGIN
          A[i,i] := H4;
          R[i,i,i] := H5;
        END;
      END;
    END;
  END;
END
```

The `ALIGN` declaration and the `ALIGNED WITH` statement result from automatic data and computation alignment that we intentionally included in the above transformation. The alignment specifications determine that the expressions `Y[2*i]`, `Y[2*i+1]`, `Y[2*i-1]`, `A[i,i]`, and `R[i,i,i]` cause remote data access during execution. Access vectorization divides these expressions into reads and writes in order to cluster remote reads at the beginning and remote writes at the end of each asynchronous block.

5.4 Process Virtualization

In general, automatic process virtualization simulates n processes on p processors where $p < n$. The most efficient technique is looping. Each of the p processors loops over $\lceil n/p \rceil$ processes. Thus, loosely synchronous and completely asynchronous parallel computations can be executed by running sequential loops on every processor. This preserves semantics because asynchronous parallelism does not prescribe any temporal ordering on the parallel computations. A sample transformation implementing the automatic virtualization of an asynchronous `FORALL` follows below.

```
FORALL i: [1..n] NOSYNC DO
  X[i] := i;
END

        ↓

TYPE PROCESSORS = [0..(p-1)];

VAR v := (n+p-1) DIV p;  (* ceil(n/p) *)

BEGIN
  FORALL k: PROCESSORS NOSYNC DO
    FOR j:=1 TO v DO
      VAR i := k*p + j;
      BEGIN
        X[i] := i;
      END;
    END;
  END;
END
```

6 Conclusion

We introduced the exciting new parallel programming language Modula-3* and showed how to compile it in a portable way. The optimizing compilation relies on Modula-3π source-to-source transformations. This offers great flexibility for future extensions, e.g. the integration and combination of new optimization techniques. As for performance, the practical quality of the restructuring optimizations will be validated by benchmark measurements on MIMD, SIMD, and SISD machines.

We have intentionally excluded exceptions in parallel computations from our discussions. The same applies to massively parallel object-oriented programming with Modula-3*. The central problems of the latter are parallel memory management and garbage collection on non-uniform memory access machines. Furthermore, some SIMD machines complicate object implementation as they do not support indirect pointer dereferencing.

A New Grammar Rules

A.1 Modula-3*

```
ForallSt ::= FORALL IdDecl {IdDecl}
             [NOSYNC] DO S END.
IdDecl   ::= IdList ":" Type.
NoSyncSt ::= NOSYNC S END.
Stmt     ::= ForallSt | NoSyncSt.
```

A.2 Modula-3π

```
Align      ::= ALIGN Expr {"," Expr}
               WITH Expr.
AlignDecl  ::= FORALL IdDecl {IdDecl}
               Align {"," Align}.
AlignedSt  ::= Alignment DO S END.
Alignment  ::= ALIGNED WITH Expr.
Alloc      ::= LOCAL | SPREAD
             | CYCLIC ["(" Expr ")"].
AllocIndex ::= Type [Alloc] {"," Type [Alloc]}
             | [Alloc].
ArrayType  ::= ARRAY AllocIndex OF Type.
             | ARRAY "(" Index ")" Alloc
               OF Type.
Decl       ::= AlignDecl.
ForallSt   ::= FORALL IdDecl {IdDecl}
               [Alignment] [NOSYNC] DO S END.
Index      ::= Type {"," Type}
             | OF ARRAY {OF ARRAY}.
Stmt       ::= AlignedSt | ForallSt.
```

References

[1] U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publ., 1988.

[2] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Language Definition. *ACM SIGPLAN Notices*, Vol. 27, No. 8, August 1992.

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical TR90-141, Dept. of Computer Science, Rice University, December 1990.

[4] G. Goff, K. Kennedy, and C.-W. Tseng. Practical Dependence Testing. CRPC Technical Report No. 90103, March 1991 (revised version).

[5] E.A. Heinz. Modula-3π Language Definition. Technical Report, Dept. of Informatics, University of Karlsruhe, 1993.

[6] E.A. Heinz. Modula-3* Language Definition. Technical Report, Dept. of Informatics, University of Karlsruhe, 1993.

[7] E.A. Heinz and M. Philippsen. Synchronization barrier elimination in synchronous FORALLs. Technical Report No. 13/93, Dept. of Informatics, University of Karlsruhe, 1993.

[8] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing '91*, Albuquerque, November 1991.

[9] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[10] P. Lukowicz. Code-Erzeugung für Modula-2* für verschiedene Maschinenarchitekturen. Master's Thesis, Department of Informatics, University of Karlsruhe, February 1992

[11] M. Philippsen and W.F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Lecture Notes in Computer Science 591, pp. 169–183, Springer Verlag, October 1991.

[12] W.F. Tichy and C. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report No. 4/90, Dept. of Informatics, University of Karlsruhe, January 1990.

[13] W.F. Tichy, M. Philippsen, E.A. Heinz, and P. Lukowicz. From Modula-2* to efficient parallel code. In *3rd Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 6–9, 1992.