The CuPit Compiler for the MasPar MP-1 and MP-2
A Literate Programming Document

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
76128 Karlsruhe, Germany
++49/721/608-4068, Fax: ++49/721/694092

January 7, 1995

Technical Report 1/95

Abstract

This document contains the complete source code of the CuPit compiler for the MasPar MP-1/MP-2 SIMD parallel machines. The compiler is presented as a FunnelWeb literate programming document that contains definitions for the various specification files needed by the Eli compiler construction system. The exactly same set of files that enabled FunnelWeb to produce this document also enable Eli to produce the complete executable compiler, run time system, and standard library. In this document the source code is complemented by interspersed documentation text and several larger introduction text blocks and appendices, in particular a description of all errors found in the compiler during its development and use. The compiler takes CuPit source code as input and produces MPL source code as output. CuPit is a special purpose language for neural network algorithms which dynamically change the topology of the neural network. The compiler is designed to optimize the irregular problems that arise when executing such algorithms for both data locality and load balancing. The compiler can produce several different versions of code: (1) a plain do-as-good-as-you-can-without-any-tricks one (unoptimized), (2) one that uses a better data distribution (statically optimized), (3) one that contains additional instructions to collect information about program behavior at run time, also known as the *rti version*, meaning "run time information version" (dynamically optimized).

Contents

PART 0: Introduction

This introduction part tells why and in which context this compiler was built, gives a rough idea of what the Eli compiler construction system is, which was used to built the compiler, gives a short introduction into the architecture and properties of the MasPar MP-1 machine and its MPL programming language, and describes how to read the rest of this document.

1 Purpose of this compiler

The language and compiler specified in this document were designed and implemented in the course of the research project leading to my Ph.D. The research goal was to find and evaluate a way to compile neural algorithms onto massively parallel machines for optimal data locality and load balance. The underlying neural networks should be allowed to vary their topology dynamically and the load balancing operations should be static in the sense that upon entry into a parallel section the work distribution for this section is fully determined.

The key idea to these goals is a particular data distribution scheme which this compiler implements prototypically. The compiler is not meant for production use but as a tool to empirically evaluate the behavior of the data distribution scheme. The compiler was written (from scratch) entirely by myself between June 1993 and April 1994.

2 The Eli compiler construction system

For those who have no knowledge of the Eli compiler construction system used for this compiler, here is a quickquickquick introduction into the major features of Eli. The compiler presented in this report was written using Eli 3.5.

Eli is a system that tries to hide from the user as many details of compiler construction as possible. The idea is to make the compiler writer express a lot of specifications from which the compiler is then constructed completely automatically. Therefore, Eli contains several special-purpose languages that allow to express specifications of the solutions for certain particular well-understood areas of compiler construction; other such areas are covered by re-usable modules. Where no such generic understanding (and thus no special-purpose language or re-usable module) is available, C code has to be supplied by the compiler writer.

Eli is centered around a tool that compiles declarative specifications of attribute grammars into a C program that is able to compute all attributes in this grammar. Eli provides special purpose languages for scanner specifiaction via regular expressions (`gla`), parser specification via LALR(1) grammars (`con`), concrete-to-abstract-syntax-mapping specification (`sym`), definition table functionality specification (`pdl`), operator identification (`oil`), attribute grammar specification (`lido`), tree-structured program text generation (`ptg`), and compiler command line processing (`clp`). In addition there are re-usable modules for symbol table handling, error message generation, consistent renaming, and several other tasks. The names given in parentheses above are also the file name extensions of the corresponding specification files; you'll find these extensions at several places in this document.

To explain all these various languages and modules would clearly lead too far in this introduction. I suggest that the reader relies on the intuitive understanding of the specifications which will be correct in most cases. Details that cannot be understood can most often be ignored without hampering overall understanding of the compiler's structure; just a few important pieces of information shall be given here:

The declaration of a property X in a `pdl` (property definition language) file induces implementations of two functions `Set`X (to set or re-set an X value) and `Get`X (to read an X value or return a default value when the property is not set).

The declaration of a program text template t in a `ptg` (program text generator) file induces the implementation of a function `PTGt` that has as many parameters as are mentioned in the template (as either $, numbered consecutively, or $1, $2, etc.). All these parameters are of type `PTGNode`, as is the function's result. An empty `PTGNode` is called `PTGNULL` (or `PTGNull()`).

In a `lido` file, a clause of the form `CONSTITUENTS Symbol.Attr WITH (Type, Merge, Create, Null)` means that from all subtrees of the current context the values of `Symbol.Attr` are collected and combined into a single value of type `Type`, where an empty `Type` object is constructed by a call to the 0-ary operation `Null`, `Attr` can be converted into a `Type` by a call to the 1-ary operation `Create`, and two `Type` objects can be merged into a single one by a call to the 2-ary operation `Merge`. An expression of the form `ORDER(A, B, C)` means that all the expressions `A, B`, and `C` are evaluated in the given order and the last one (`C` in this case) is returned as the result. An arbitrary number of expressions can be given in an `ORDER` expression.

The `head` files are technical details you may ignore. The `tplr` files introduced in the first part of code generation contain C with refinements. They are converted into plain C with the tool C-Refine before they are used — then called `tpl` files.

3 The MasPar MP-1/MP-2 computers

For those who have no knowledge of the MasPar machines but want to understand the code in this document, here is a quickquick introduction into the machine architecture and the MPL programming language.

3.1 Machine architecture

The MasPar MP-1 and MP-2 machines are massively parallel SIMD computers. They have 1024 or 2048 or ... or 16384 simple 4-bit wide processors called "processing elements" or PEs for short. Each PE has the same amount of memory which is either 16 KB or 64 KB. The machine for which the compiler is meant to produce code in the context of this project is a MP-1216A, an MP-1 with 16384 PEs with 16 KB memory each. The PEs are arranged as a 2D toroidal mesh and numbered row-wise with the upper left corner being PE number 0. Compass points are used to describe directions in the PE array with up being north and left being west.

The PEs are controled by a more powerful central processor called the ACU, which is 32 bit wide. The programs run on the ACU and are started there from a front-end computer which is a DEC 5000 workstation.

The MasPar has two independent interprocessor communication networks: The "global router" (or just "router") allows for arbitrary communication patterns with automatic resolution of multiple reads and automatic random resolution of multiple writes. The latency is about 230 microseconds, bandwith is about 250 kilobit per second per PE. Both of these values are for a random permutation, which internally takes about 48 communication steps. For simpler communication patterns, latency and bandwith can be significantly higher. A particular case of a "simpler pattern" is when not all PEs are participating — when PE activity is not clustered, communication performance increases about linearly with sinking PE activity.

The "X-net" is a simpler communication network. In an xnet communication all participating PEs communicate with a partner in the same direction (the hardware supports only the eight main compass directions N,NE,E,SE,S,SW,W,NW, but the library also allows other directions). All these partners must also be the same distance away. The advantage of the xnet is that xnet communications are blindingly fast (roughly operand size times distance times 13 microseconds, with the operand size measured in bits).

Since both communication networks are line-switched, a fetch is not more expensive than a send.

With all PEs participating a 32 bit random router permutation communication costs about as much as 137 integer additions, 14 integer multiplications, 12 float additions, 8 float multiplications, or 43 loads. A nearest neigbor xnet communication costs only half as much as a load, a distance 64 xnet communication costs about as much as 30 loads.

3.2 MPL programming language

The MasPar machines are programmed in a data-parallel variant of C, called MPL (the "MasPar programming language"). Its main idea is to use so-called "plural variables" to express parallelism: a variable declared plural, e.g. `plural int a`, is allocated once on each PE, each of these exemplars may hold a different value. Operations on this variable occur on each PE, thus implementing data-parallelism. Data that is non-plural is also called `singular` and is allocated on the ACU.

The notion of plural variables extends to control flow in a natural way: An `if` with a plural condition will in effect execute its `then` part on all PEs where the condition is true and the `else` part on all others that were active during the evaluation of the `if` condition. A plural `while` may lose the participation of some of its PEs before each iteration; it terminates as soon as no more PEs participate. All such control flow constructs establish the so-called "active set" of PEs that participate in the execution of the next statement(s). The active set is always a subset of the active set in the next "surrounding" (in a dynamic sense) control flow construct. This establishes a stack of active sets; upon exit from the body of a control flow construct the PEs of the previous active set are activated. A special case is the `all` statement: Upon entry into its body, all PEs are active, independent of the surrounding active set.

The communication networks are used as follows: The plural expression `router[pe].expr` evaluates the plural expression `expr` on the PEs given by the plural expression `pe` and returns it on the PE that executed the router expression. `expr` must evaluate into a single integer, pointer, or floating point value; it is not possible to communicate records or arrays as a whole with the `router` expression. Note that `expr` is used *by name*, i.e., in `router[pe].a[i]`, the value of `i` is also determined on the remote PE!

Similarly, the plural expression `xnetE[dist].expr` evaluates the plural expression `expr` on the PEs that are `dist` PEs to the right (with toroidal wraparound). The `xnetE` can be replaced by `xnetSW` to access PEs in right-down direction etc. The expression `dist` must be singular integer.

In addition you can access a plural value on a single PE from the ACU using the singular expression `proc[pe].expr` (here, `pe` must be singular), which evaluates the plural expression `expr` on only the PE `pe` and returns the result.

Router, xnet, and proc expressions can also be used on the left hand side of assignments. It is possible to communicate with PEs that are not currently active.

A bit difficult to understand are declarations of pointers: An `X*` always points into ACU memory (i.e. to non-plural variables). A `plural X*` points into PE memory (i.e., to a plural variable), but the pointer itself is singular and thus always points to only one and the same variable on all PEs. A `plural X* plural` points into PE memory (i.e., to plural variables) and is also itself plural so that it may point to a different variable on each PE. Nevertheless, such plural pointers to plural data can only point to data located on the same PE. (In the compiler, we introduce a type called `Gptr` to implement fully global addresses). Note that singular pointers are significantly more efficient than plural pointers.

3.3 MPL library

The MPL library implements many functions that directly correspond to functions of the standard C library but have plural operands and/or results. These have names just like their sequential counterparts with an additional `p_` prepended.

What is more interesting are the communication functions: `sp_rsend (plural int target_pe, plural char *src, plural char *plural dest, int bytes)` sends `bytes` bytes beginning at the local address `src` to the address `dest` on the remote PE indicated by its number `target`. This is a functionality similar to that of the `router` expression; the main differences are: (1) you don't have to give an expression that is evaluated by name but can give a direct address instead and (2) you are not limited to individual values but can send whole records or arrays at once. For `sp_rsend`, the `src` must be a singular pointer while the `dest` must be a plural pointer; this is indicated by the two prefix characters `sp`. The versions `ss, ps`, and `pp` of `rsend` are available as well as are the corresponding `rfetch` to fetch data from `src` into `dest`.

Along the same lines, there is a set of 8 functions for xnet communication called `ss_xsend` to `pp_xfetch`. These functions extend the 8 native directions available for xnet to arbitrary offsets in the left/right direction (or west/east-direction or x-direction) and the up/down direction (or north/south-direction or y-direction), e.g. `ss_xsend(int dy, int dx, plural char* src, plural char* dest, int bytes)`. Note that the first argument is `dy`, not `dx`.

In addition, there is a large number of reduction operations that combine a value from all PEs into a singular value, e.g. `unsigned short reduceAdd16u (plural unsigned short x) Adds the 16-bit` unsigned values `x` from all active PEs and returns the result. Corresponding functions are available for Multiplication, Minimum, Maximum, and bitwise operations for all signed and unsigned 8, 16, 32, and 64 bit integer as well as float and double values. (The 64 bit integer type is called `long long`)

4 How to use this document

This document is structured along the compiler phases and the CuPit language constructs. It is not meant to be read completely but should serve as a reference to investigate about individual aspects of the compiler that are of interest. To find the parts of the document relevant to such an aspect of interest the reader should start with an examination of the table of contents.

The document is written as a "web" using a literate programming tool called FunnelWeb[Wil92], which is a program designed after Donald Knuth's idea of literate programming and is able to produce arbitrarily many files (not only one, as Knuth's original WEB program) in any "language" (not only in Pascal, as Knuth's original WEB program). The idea is that typeset documentation text and program code can be combined in a single document and the code can also be split into arbitrary parts, which may be arranged in any order so that you can chose the order that is most appropriate to write or understand the code instead of the order required by your language(s) and programming environment. These parts of code are usually called "chunks" in the literate programming community; FunnelWeb uses the term "macros". Macros are either used by other macros or are attached to an output file, i.e, running FunnelWeb produces this output file.

The document consists of five main parts: Part I contains in its code part a specification of the concrete syntax of CuPit and in its text part a (rather informal) language definition[1]. Part II contains the semantic analysis of the compiler. Part III and IV contain the code generation, where part III is an introduction into the design of the code generation (including a rough description of the key ideas used in the compiler) plus some "infrastructure" (large code templates), while part IV contains the actual code generation itself. Part V contains the run-time system of the compiler. Parts I, II, and IV are all substructured in the same way along the language constructs of CuPit. Before these five main parts there is the introduction you are currently reading. After these five main parts follows a part VI of miscellaneous compiler fragments not directly belonging somwhere else, a few appendices (in particular a list of compiler limitations), a short bibliography, and the keyword index.

This structure should allow to access for reading any part of the compiler relatively easily. Where additional access information is needed, the index should be consulted. It contains entries for most terms and identifiers that are used not only in a local fashion. The FunnelWeb macros are also indexed. The entries in the index are labeled by page number, where an entry in italic font means that the corresponding term or identifier is *defined* on that page. Entries that begin with a dot (such as file name suffixes) or an underscore (such as several type names) are given *without* these characters in the index. Entries of generic identifiers that have the form `N_T` where `N` is the stem of the identifier and `T` is for instance a type

[1] This part, together with some additional sections and a tutorial is also used to construct another document: The CuPit language reference manual. Due to this fact, some of the statements in this part are not quite true since they describe the language as such while the concrete implementation has some restrictions.

name (where there is one such identifier for any one type name from a certain set of typenames) are often given in the index as `N` only or as `N_X` (where `X` is really just an X), whichever seemed more intuitive. The names of all FunnelWeb macros are also mentioned in the index. The macros are numbered consecutively independent of page numbers, the index entry of a macro is nevertheless its page number. Note that the page numbers given in the index entries of things defined within the body of a FunnelWeb macro refer to the beginning or end of the macro, so they can sometimes be off by one or two pages.

I do not claim that this document is a detailed or even excellent description of my compiler's code in the sense of literate programming. But I believe that it is a much better documentation than is usually available for research prototypes.

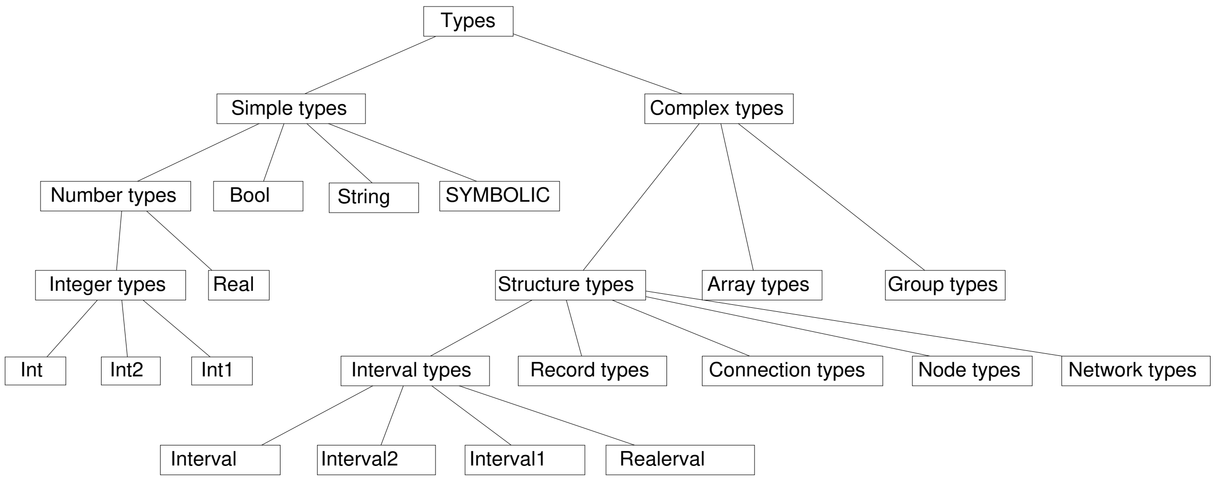


Figure 1: **CuPit type taxonomy**

PART I: Syntax and Language Definition

The syntax description consists of the context free grammar (LALR(1)) describing the concrete syntax of the language and of the scanner description describing the structure of the nonliteral nonterminal symbols.

The description of the semantics of these language constructs (as viewed by a programmer) is embedded into the syntax description; this part of the compiler document is the very same that is used for the core part of the CuPit language reference manual, where you can also find a tutorial example of CuPit programming [Pre94].

5 Type definitions

5.1 Overview

There are several categories of types in CuPit:

1. Elementary types, such as `Int`, `Real`, `Bool`, `String`, and enumerations (`SYMBOLIC` types).
2. Intervals of `Int` or `Real`.
3. Record types.
4. Connection types.
5. Node types.
6. Network types.
7. Arrays of objects of simple types, intervals, record types, node types, or array types.
8. Groups of objects of node types (i.e. collections with or without fixed order)

The elementary types are also called *simple types*. The elementary types except `Bool`, `String`, and enumerations are called *number types*. The interval, record, connection, node, and network types are also called *structure types*. The structure types, interval types, array types, and group types are called *complex types*. You can see the whole taxonomy of the CuPit type system in figure 1.

This is the general syntax of type definitions:

Type Definition[1] $\equiv$ 1

```
{
 TypeDef:
```

```
    'TYPE' NewTypeId 'IS' TypeDefBody 'END' OptTYPE.

NewTypeId:
    UppercaseIdent.

TypeDefBody:
    SymbolicTypeDef /
    RecordTypeDef /
    NodeTypeDef /
    ConnectionTypeDef /
    ArrayTypeDef /
    GroupTypeDef /
    NetworkTypeDef.

OptTYPE:
    /* nothing */ /
    'TYPE'.
```

Symbolic Type Definition[2]
Record Type Definition[3]
Node Type Definition[6]
Connection Type Definition[8]
Array Type Definition[9]
Group Type Definition[10]
Network Type Definition[11]
}

This macro is invoked in definition 55.

A type definition may appear only on the outermost level of a CuPit program (i.e. not within procedures). The `TypeId` mentioned in the `TypeDef` is introduced as a new type name and bound to the definition given in the `TypeDefBody`. The new `TypeId` is defined and visible in the rest of the program after the point where it appears first in its own definition, i.e., types must be defined before they can be used in the definition of another type or in the definition of an object. Type names must not be redefined.

All types that occur in a CuPit program have an explicit name and two types are identical only if they have the same name. *Design rationale:* This makes the semantics of the language much simpler.

The individual kinds of definitions will be explained in the next few subsections.

5.2 Simple types

Among the basic types of CuPit are truth values `Bool`, integral numbers `Int` and floating point numbers `Real`. The exact representation and operation semantics of these types is machine dependent. There are three variants of `Int`, namely `Int1`, `Int2`, and `Int`. These are one-byte, two-byte, and four-byte signed integers, respectively.

Other simple types are the `String` type, which represents pointers to arrays of bytes terminated by a byte with value 0 (like in C), and the so-called `SYMBOLIC` types, which are defined by giving a list of names that represent the set of values of the type. Thus a `SYMBOLIC` type is similar to an enumeration type in MODULA-2 or in C, except that in CuPit symbolic values are not ordered and cannot be converted into or created from integer values. The only operations that are defined on symbolic types are assignment and test for equality.

Objects of simple types may occur as members in any other type, as global variables and as local variables in all kinds of procedures and functions.

2 *Symbolic Type Definition*[2] ≡
{

```
SymbolicTypeDef:
   'SYMBOLIC' NewEnumIdList OptSEMICOLON.

OptSEMICOLON:
   /* nothing */ /
   ';'.

NewEnumIdList:
   NewEnumId /
   NewEnumIdList ',' NewEnumId.

NewEnumId:
   LowercaseIdent.
}
```

This macro is invoked in definition 1.

5.3 Interval types

Types can be defined that can hold two integer or real values and mean the compact integer or real interval between the two. Objects of interval types may occur as elements of any complex type, as global variables and as local variables in all kinds of procedures and functions.

Objects of type `Interval`, `Interval1`, and `Interval2`, use objects of type `Int`, `Int1`, `Int2` respectively, to represent their current maximum and minimum; `Realerval` objects use `Real` values. The strange name `Realerval` is just a play on words.

Design rationale: The reason for introducing `Interval` types explicitly in the language is that some special operations shall be defined for them.

5.4 Record types

Records are compounds of several data *elements* (also called *components* or *fields*) and are similar to RECORDs in Modula-2 or structs in C. Records types consist of internal data elements and a number of operations, which can be performed on them, the so-called *record procedures* (and record functions).

Objects of record types may occur as elements in any other complex type, as global variables, and as local variables in all kinds of procedures and functions.

Record Type Definition[3] ≡ 3

```
{
RecordTypeDef:
   'RECORD' RecordElemDefList.

RecordElemDefList:
   RecordElemDef ';' /
   RecordElemDefList RecordElemDef ';'.

RecordElemDef:
   RecordDataElemDef /
   MergeProcDef /
   ObjProcedureDef /
   ObjFunctionDef.
}
```

This macro is defined in definitions 3 and 4.
This macro is invoked in definition 1.

For the meaning and restrictions of procedure and function definitions in records, see section 7. The data element definition will be explained now:

4 *Record Type Definition*[4] ≡

```
{
 RecordDataElemDef:
    TypeId InitElemIdList.

 InitElemIdList:
    InitElemId /
    InitElemIdList ',' InitElemId.

 InitElemId:
    NewElemId /
    NewElemId ':=' Expression.

 NewElemId:
    LowercaseIdent.
```

Type Identifier[5]
}

This macro is defined in definitions 3 and 4.
This macro is invoked in definition 1.

5 *Type Identifier*[5] ≡

```
{
 TypeId:
    UppercaseIdent.
}
```

This macro is invoked in definition 4.

Each name in the initialized-identifier list introduces an element of the record in the sense of a record field in Modula-2 or a component of a struct in C. The name of the element is local to the record, i.e., the same name may be used again as the name of a procedure or data object or as the name of an element in a different structure type.

Elements of records may be of simple type, interval type, record type, or array of those. Initializers for individual elements in a record type may be given. The meaning of an initializer `x` at an element `c` is that for each object A of the record type the element `c` of this object is initialized to the value of expression `x` upon creation of that object A. The initializer may consist of any expression of objects visible at that point in the program. The type of the expression must be compatible to the type of the element.

Other initializers for elements of the record type may exist in the object declarations using the record type or in the declarations of types that contain elements of the record type. These initializers apply later and thus overwrite the effect of the initializers here.

Example:

```
TYPE Atype IS RECORD  Int a, b = 7;  END
TYPE Btype IS RECORD  A x = A (2, 5);
                      Int c = 0;     END
```

Here, element `b` will be initialized to 7 in an `Atype` object, while element `x.b` will be initialized to 5 in an `Btype` object. Element `a` will be initialized to an undefined value in an `Atype` object, because no initializer is given. Programs that rely on certain values in such undefined objects are erroneous.

5.5 Node types

The nodes are the active elements of neural computation (some people call them *units* or even *neurons*). In CuPit, Nodes consist of input and output interface elements, internal data elements, and a number of operations, the so-called *node procedures*, that operate on the internal data elements and the connections attached to the interface elements.

Objects of node types may only occur as members in objects of group types and array types. They are not allowed as global variables or as local variables or parameters in any kind of procedure or function.

Node Type Definition[6] ≡ 6

```
{
 NodeTypeDef:
    'NODE' NodeElemDefList.

 NodeElemDefList:
    NodeElemDef ';' /
    NodeElemDefList NodeElemDef ';'.

 NodeElemDef:
    NodeInterfaceElemDef /
    NodeDataElemDef /
    MergeProcDef /
    ObjProcedureDef /
    ObjFunctionDef.

 NodeDataElemDef:
    TypeId InitElemIdList.
}
```

This macro is defined in definitions 6 and 7.
This macro is invoked in definition 1.

For the meaning and restrictions of procedure and function definitions and merge procedure definitions in nodes, see section 7. The data element definitions are analogous to those in record types and obey the same rules. The node interface element definitions will be explained now:

Node Type Definition[7] ≡ 7

```
{
 NodeInterfaceElemDef:
    InterfaceMode TypeId InterfaceIdList.

 InterfaceMode:
    'IN' /
    'OUT'.

 InterfaceIdList:
    NewInterfaceId /
    InterfaceIdList ',' NewInterfaceId.

 NewInterfaceId:
    LowercaseIdent.
}
```

This macro is defined in definitions 6 and 7.
This macro is invoked in definition 1.

Interface elements are no data elements but instead have the property that connections can be attached to them. The type name given in an interface element definition must be the name of a connection type; only connections of this type can be attached to the interface element. For interface mode `IN`, the connections are incoming connections: the output of these connections is connected to the interface element. For interface mode `OUT` the connections are outgoing connections: the input of these connections is connected to the interface element. The name of an interface element of a particular node object stands for all the connections that are attached to that interface element at once. The visibility of the name of an interface element obeys the same rules as the visibility of the name of a data element.

It is allowed to have several interface elements with the same interface mode in a single node type. Initializers for interface elements cannot be given in a node type declaration.

5.6 Connection types

Connections are the communication paths along which data flows from one node to another in a network. A connection object may contain arbitrary data and may perform arbitrary operations on it.

Objects of connection types cannot be declared explicitly; they may occur only implicitly connected to an output interface element of one node and to an input interface element of another (maybe the same) node. They are not allowed as members in any other type, nor as global variables nor as local variables in any kind of procedure or function. They can, however, be passed as parameters to external functions.

Connections are directed, i.e., they are not connections *between A and B*, but either *from A to B* or *from B to A*. Nevertheless, data can be transported along a connection in both directions. *Design rationale:* Connections must be directed because otherwise it is very difficult to provide an efficient implementation: Without direction it is not possible to store the actual connection data always at, say, the input end of the connection; thus we could not achieve data locality between connections and (at least one of the two) attached nodes.

8 *Connection Type Definition*[8] ≡

```
{
 ConnectionTypeDef:
    'CONNECTION' ConElemDefList.

 ConElemDefList:
    ConElemDef ';' /
    ConElemDefList ConElemDef ';'.

 ConElemDef:
    ConDataElemDef /
    MergeProcDef /
    ObjProcedureDef /
    ObjFunctionDef.

 ConDataElemDef:
    TypeId InitElemIdList.

}
```

This macro is invoked in definition 1.

For the meaning and restrictions of procedure and function definitions and merge procedure definitions in connections, see section 7. The data element definitions are analogous to those in record and node types and obey the same rules.

5.7 Array types

Arrays are linear arrangements of several data elements of the same type (called the *base type* of the array). The number of data elements in the array is called the *size* of the array. The elements can be accessed individually by means of an *index* as known from Modula-2. The lowest index to an array is always 0, the highest index is the size of the array minus one. An attempt to access an array using a negative index or an index that is too large is a run-time error.

Objects of array types may be used wherever objects of their element type may be used.

Array Type Definition[9] ≡ 9

```
{
ArrayTypeDef:
   'ARRAY' '[' ArraySize ']' 'OF' TypeId.

ArraySize:
   Expression.
}
```

This macro is invoked in definition 1.

The array size expression must be of integer type and must contain only constant values, so that it can be evaluated at compile time. The value of the expression determines the size of the array.

5.8 Group types

Groups are linear arrangements of several data elements of the same type (called the *base type* of the group). The number of data elements in the group is called the *size* of the group. The elements can be accessed individually by means of an *index* just like for an array. The lowest index to a group is always 0, the highest is the size of the group minus one. An attempt to access a group using a negative index or an index that is too large results in a run-time error.

Objects of group types may occur only as elements of network types. The base type of a group type must be a node type.

This far, groups and arrays are mostly the same. The main difference between groups and arrays is that groups are dynamic in size: There are operations to add new elements to a group at the end of the current index range, or to delete elements from the end of the current index range (see section 8.8). These operations cause the size of the group to change, but keep the indices of those elements of the group constant that already existed before the operation and still exist after it. In contrast to these operations there are others, which also change the size of the group, but do *not* necessarily leave the indices of constantly existing elements unchanged: Elements of a group can self-delete, even if they are not the last ones of the group and elements of a group can self-replicate (i.e. make one or several additional copies of themselves), even if they are not the last element of the group (see section 8.8). Such operations cause the indices of all the elements of the group to be recomputed. For arrays, the identity of an element with index `i` remains constant for the whole lifetime of the element. This is not true for groups: A constant index `i` is not guaranteed to refer to the same object in a group after a self-delete or self-replicate operation has been performed on the group (see section 8.8). The initial size of a group is 0.

Group Type Definition[10] ≡ 10

```
{
GroupTypeDef:
   'GROUP' 'OF' TypeId.
}
```

This macro is invoked in definition 1.

5.9 Network types

Networks are the central data structures of neural algorithms. A network contains one or more groups or arrays of nodes, which are interconnected by connections. Other data may also be present in a network. Objects of network types may occur only as global variables.

Network Type Definition[11] ≡ 11
```
{
 NetworkTypeDef:
    'NETWORK' NetElemDefList.

 NetElemDefList:
    NetElemDef ';' /
    NetElemDefList NetElemDef ';'.

 NetElemDef:
    NetDataElemDef /
    MergeProcDef /
    ObjProcedureDef /
    ObjFunctionDef.

 NetDataElemDef:
    TypeId InitElemIdList.
 }
```
This macro is defined in definitions 11.
This macro is invoked in definition 1.

The data element definitions for a network type are similar to those of a node type, except that arrays and groups of nodes are allowed as elements additionally. Arrays and groups cannot be initialized explicitly. Nodes can be used as elements of a network only in groups or arrays — individual nodes are not allowed.

6 Data object definitions

12 *Data Object Definition*[12] ≡
```
{
 DataObjectDef:
    TypeId AccessType InitDataIdList.

 AccessType:
    'CONST' /
    'VAR' /
    'IO'.

 InitDataIdList:
    InitDataId /
    InitDataIdList ',' InitDataId.

 InitDataId:
    NewDataId /
    NewDataId ':=' Expression.

 NewDataId:
    LowercaseIdent.
 }
```
This macro is invoked in definition 55.

Objects can be defined as either constants or variables or I/O areas. The only difference between constants and variables is that constants *must* be initialized and cannot be assigned to at any other point in the source code. It is possible, though, that a constant is not really allocated in memory as a data object at run-time when its properties are completely known at compile-time.

The I/O area data object category is CuPit's way to handle input and output. The exact layout and handling of I/O area objects are machine-dependent and must be specified separately for each compiler. *Design rationale:* Since the semantics of actual parallel I/O are tricky, CuPit defines only buffer operations and leaves the actual transfer of these buffers to machine-dependent external procedures.

An I/O area is a data object that is used to move data into a CuPit program from and out of a CuPit program to an external program part. Defining an I/O area basically means to declare a name for a variable whose storage must be allocated by an external program part and whose memory layout is defined by each CuPit compiler in a target machine dependent way. I/O areas can be used as arguments to external functions and special CuPit operators exist to move data from an I/O area into a group or array of nodes or vice versa (see section 8.3). I/O areas are allowed to occur everywhere. However, in network or node or connection procedures they are usually useless.

7 Subroutine definitions

7.1 Overview

There are several types of subroutines in CuPit:

1. Procedures.
2. Functions.
3. Object procedures and object functions, which are much like normal procedures and functions.
4. Reduction functions, to combine many values into one.
5. Winner-takes-all functions, to reduce a parallel context into a sequential one.
6. Object merge procedures, to unite multiple replicates of a data object into one.

We will explain each of these types in order.

Subroutine Definition[13] ≡ 13

{
Procedure Definition[14]
Function Definition[15]
Reduction Function Definition[16]
Winner-takes-all Function Definition[17]
Object Merge Procedure Definition[18]
Statements[19]
}

This macro is invoked in definition 55.

7.2 Procedures and functions

Procedure Definition[14] ≡ 14

{

```
ProcedureDef:
   'PROCEDURE' NewProcedureId SubroutineDescription OptPROCEDURE.

NewProcedureId:
  LowercaseIdent.

SubroutineDescription:
   ParamList 'IS' SubroutineBody 'END' /
   ParamList 'IS' 'EXTERNAL'.
```

```
ParamList:
   '(' ')' /
   '(' Parameters ')'.

Parameters:
   ParamsDef /
   Parameters ';' ParamsDef.

ParamsDef:
   TypeId AccessType ParamIdList.

ParamIdList:
   NewParamId /
   ParamIdList ',' NewParamId.

NewParamId:
   LowercaseIdent.

SubroutineBody:
   Statements.

OptPROCEDURE:
   /* nothing */ /
   'PROCEDURE'.

ObjProcedureDef:
   'PROCEDURE' NewObjProcedureId SubroutineDescription OptPROCEDURE.

NewObjProcedureId:
   LowercaseIdent.
}
```

This macro is invoked in definition 13.

The semantics of a procedure definition is similar to that of a procedure definition in Modula-2:
`PROCEDURE p (CONST T1 a, b; VAR T2 c) IS stmts END`
defines a procedure with the name `p` with three parameters. The parameters `a` and `b` have type `T1` and are available in the body of the procedure just like constants of same name and type, i.e. they may be read but not assigned to. Parameter `c` has type `T2` and is available in the body of the procedure just like a variable of same name and type. The body of the procedure consists of `stmts`.

If the procedure definition is part of the definition of a record type, node type, connection type, or network type, the procedure is called an *object procedure*. In this case, the object for which the procedure has been called is visible as `ME` in the procedure body. All elements of that object are visible and can be accessed using the selection syntax (e.g. `ME.a` to access a element `a`). `VAR` parameters are allowed for an object procedure only when the procedure is only called from other object subroutines of the same type. Otherwise, object procedure definitions are just like normal procedure definitions.

If the procedure body is replaced by the `EXTERNAL` keyword, the procedure is only declared, but not defined and must be implemented externally. *Design rationale:* The purpose of an `EXTERNAL` procedure definition is to make procedures and their parameter lists visible, so that a CuPit program can call them.

Parameters of node or connection types are allowed for external procedures only.

15 *Function Definition*[15] $\equiv$

```
{
FunctionDef:
   TypeId 'FUNCTION' NewFunctionId SubroutineDescription OptFUNCTION.
```

```
NewFunctionId:
   LowercaseIdent.

OptFUNCTION:
   /* nothing */ /
   'FUNCTION'.

ObjFunctionDef:
   TypeId 'FUNCTION' NewObjFunctionId SubroutineDescription OptFUNCTION.

NewObjFunctionId:
   LowercaseIdent.
}
```

This macro is invoked in definition 13.

The semantics of a function definition is analogous to that of a procedure definition. The difference is that for a function a return type has to be declared. The value is returned in the function body using the `RETURN` statement with an expression. Connection, node, and network types are not allowed as return types of functions. An object function definition looks exactly like a normal function definition. The only difference is that for an object function definition the `ME` object that denotes the object the function was called for is visible in the body; neither `ME` nor its elements can be changed. No function may have a `VAR` parameter.

7.3 Reduction functions

Reduction Function Definition[16] ≡ 16

```
{
ReductionFunctionDef:
   TypeId 'REDUCTION' NewReductionFunctionId 'IS'
      ReductionFunctionBody 'END' OptREDUCTION.

NewReductionFunctionId:
   LowercaseIdent.

ReductionFunctionBody:
   Statements.

OptREDUCTION:
   /* nothing */ /
   'REDUCTION'.
}
```

This macro is invoked in definition 13.

The definition of a reduction function introduces a binary operator (which must be commutative and associative). This operator is used to reduce multitudes to single values in an implicit way.
For a declaration `T REDUCTION op IS body END`, the objects `ME` and `YOU` are implicitly declared as `T CONST` and are visible in `body`. `T` is the type of the values that can be reduced by this reduction function.

Reduction functions can be declared only globally (i.e. outside of type definitions and procedure definitions) and are used in three different contexts: First, in a node subroutine to reduce the values delivered to a node by the set of connections attached to a single connection interface; second, in a network subroutine to reduce the values of a particular data element of all nodes of a single node group or node array, and third, in a global subroutine to reduce the values of a particular data element of all replicates of a single network.

Design rationale: A reduction function declaration can be used to construct an efficient reduction procedure that runs in logarithmic time on a parallel machine and uses knowledge about the specific data distribution in order to avoid communication operations.

Example: Given the definition
`Real REDUCTION sum IS RETURN (a+b) END`
then in a node procedure of a node type having a connection interface `in` of a connection type having a `Real` data element `val`, the statement
`REDUCTION ME.in[].val:sum INTO inSum;`
means to apply the `sum` reduction to the `val` fields of all connections attached to the `in` interface. Assuming that there are exactly three connections whose `val` values are `x, y, z`, respectively. The value of `inSum` after the statement will be either `(x+y)+z` or `x+(y+z)` or `(x+z)+y` or any commutation of one of these.

7.4 Winner-takes-all functions

17 *Winner-takes-all Function Definition*[17] ≡

```
{
 WtaFunctionDef:
    TypeId 'WTA' NewWtaFunctionId 'IS' WtaFunctionBody 'END' OptWTA.

 NewWtaFunctionId:
    LowercaseIdent.

 WtaFunctionBody:
    Statements.

 OptWTA:
    /* nothing */ /
    'WTA'.
 }
```

This macro is invoked in definition 13.

The definition of a winner-takes-all function introduces a binary operator. This operator is a comparison operator and is used to induce an ordering on the type for which the operator is defined.

For a declaration `T WTA op IS body END`, the objects `ME` and `YOU` are implicitly declared as `T CONST` and are visible in `body`. The `body` must return a `Bool` result; `true` means that `ME` is above `YOU` in the ordering defined by the operator and `false` means that it is not.

Winner-takes-all functions can be declared only globally (i.e. outside of type definitions) and are used in three different contexts: First, to select one connection per node from the sets of connections attached to a certain node interface of each node in a group of nodes; second, to select one node from a group of nodes; and third, to select a network from a set of replicated networks. See section 8.6.

Design rationale: The winner of a winner-takes-all call is always unique. Thus it is not easily possible to emulate a winner-takes-all function by a reduction and subsequent rebroadcast of the result, because the winning *value* need not be unique. A winner-takes-all function declaration can be used to construct an efficient reduction procedure that runs in logarithmic time on a parallel machine and uses knowledge about the specific data distribution in order to avoid communication operations.

7.5 Merge procedures

18 *Object Merge Procedure Definition*[18] ≡

```
{
 MergeProcDef:
    'MERGE' 'IS' MergeProcedureBody 'END' OptMERGE.
```

```
MergeProcedureBody:
   Statements.

OptMERGE:
   /* nothing */ /
   'MERGE'.
}
```

This macro is invoked in definition 13.

Merge procedures are similar to reduction functions; they also perform a reduction. Their purpose is to reunite replicated exemplars of networks or individual network elements. For a description of network replication, see section 8.8.

While replication and subsequent merging can only be *executed* for a whole network, merging is *defined* in network types, node types, and connection types separately. This way, the knowledge about how merging works for particular object types remains local to the definitions of these types.

When network merging is called, each merge procedure of the network elements is implicitly called as an object procedure, i.e., the object for which it has been called is available as `ME`. This object is also where the result of the merging has to be placed by the merge procedure. The object to be merged into `ME` is available as `YOU` with `CONST` access, i.e., writing to elements of `YOU` is not allowed. The task of the merge procedure body is to construct in `ME` the reunion of `ME` and `YOU`. A merge procedure of a network type should merge all relevant non-node elements of the network. The node elements are merged one-by-one by the respective merge procedures of the node types. A merge procedure of a node type should merge all relevant non-interface elements of the node. The connections attached to the node are merged one-by-one by the respective merge procedures of the connection types.

If no merge procedure is defined for a particular type, no merging occurs and the reunited exemplar of each object of this type is identical to a random one of the replicated exemplars of the object. *Design rationale:* Often, most of the data structures do not really need to be merged in neural algorithms.

However, merging is still performed on the enclosed parts of the data structure. That is, if no merge procedure for a network is defined, merging can still occur for the nodes and connections of this network, if merging procedures for them are defined. If no merge procedure for a node is defined, merging can still occur for its connections. *Open question: Do we need the capability to declare multiple merge procedures for the same type ?*

8 Statements

8.1 Overview

The statements available in CuPit can be divided into the following groups:

1. Statements that are common in sequential procedural languages, such as assignment, control flow, procedure call.
2. Statements that imply parallelism, such as group procedure call and reductions.
3. Statements that modify the number of data objects, i.e., create new objects or delete existing ones.

Each list of statements can have some data object definitions at its beginning. The objects declared this way are visible only locally in the list of statements. They are created at run-time just before the list is executed and vanish as soon as the execution of the list is over. This introduces a kind of block structure for local data objects into CuPit that is similar to that of C.

Statements[19] ≡ 19

```
{
 Statements:
    DataObjectDefList StatementList.
```

```
DataObjectDefList:
   /* nothing */ /
   DataObjectDefList DataObjectDef ';'.

StatementList:
   /* nothing */ /
   StatementList Statement ';'.
}
```

This macro is invoked in definition 13.

20 *Statement*[20] ≡

```
{
Statement:
   Assignment /
   InputAssignment /
   OutputAssignment /
   ProcedureCall /
   ObjectProcedureCall /
   MultiObjectProcedureCall /
   ReductionStmt /
   WtaStmt /
   ReturnStmt /
   IfStmt /
   LoopStmt /
   BreakStmt /
   DataAllocationStmt /
   MergeStmt.
```

Assignment[21]
I/O Assignment[22]
Procedure Call[23]
Reduction Statement[26]
Wta Statement[27]
Return Statement[28]
If Statement[29]
Loop Statement[30]
Break Statement[31]
Data Allocation Statement[32]
Merge Statement[33]
}

This macro is invoked in definition 55.

All these kinds of statements will now be explained individually.

8.2 Assignment

21 *Assignment*[21] ≡

```
{
Assignment:
   Object AssignOperator Expression.

AssignOperator:
   ':=' / '+=' / '-=' / '*=' / '/=' / '%='.
}
```

This macro is invoked in definition 20.

The assignment `a := b` stores a new value (as given by the expression `b`) into a data object (`a`, in this case). The types of `a` and `b` must be compatible (see section 9.2) and `b` is converted into the type of `a` if necessary. The computation of the memory location to store to (the address of `a`) may involve the evaluation of expressions, too. In this case, the the left hand side and the right hand side are evaluated in undefined order (e.g. in parallel). The assignments `a += b, a -= b, a *= b, a /= b, a %= b` have the same meaning as `a = a+b, a = a-b, a = a*b, a = a/b, a = a%b`, except that any expressions involved in computing the address of `a` are evaluated only once.

The assignment to a node object N is defined only, when this node N does not yet have any connections attached to it.

Design rationale: This is because assignment to node objects is intended to be used for initialization only. During the rest of the program run, nodes should only be changed by themselves by means of node procedures.

8.3 I/O assignment

I/O Assignment[22] ≡ 22

```
{
 InputAssignment:
    Object '<--' Object.

 OutputAssignment:
    Object '-->' Object.
}
```

This macro is invoked in definition 20.

The purpose of these special assignments is to provide a way of communication between a network and the "outer world": Since the mapping of nodes onto processors is completely left to (and known only by) the compiler, external procedures can not directly read data from nodes or write data into nodes. On the other hand, the memory mapping of I/O areas is statically defined by any compiler (see sections 6 and D), so that external procedures can easily access them for reading and writing.

The object on the left hand side must be a data element of a group of nodes (i.e., a parallel variable), the object on the right hand side must be a global `X IO`, where `X` is the type of the data element field of the nodes mentioned on the left hand side.

A single input or output assignment statement provides one value for each node of the group in each of the replicates of the respective network. For the input assignment each such value is copied from the I/O area into the data element of the appropriate node according to the I/O area data layout defined for the particular compiler. For the output assignment the value is copied from the data elements of the nodes into the I/O area.

Input and output assignments are allowed in the central agent only. *Design rationale:* The central agent is conceptually the only part of the program where knowledge about network replication is present. Since input and output assignments work on all replicates at once and the program who fills or reads an I/O area must know that; the central agent is the only program part where input and output assignments make sense.

8.4 Procedure call

Procedure Call[23] ≡ 23

```
{
 ProcedureCall:
    ProcedureId '(' ArgumentList ')'.
```

```
ProcedureId:
   LowercaseIdent.

ArgumentList:
   /* nothing*/ /
   ExprList.
}
```

This macro is defined in definitions 23, 24, and 25.
This macro is invoked in definition 20.

The semantics of a procedure call are similar to that known in languages such as Modula-2 or C: First, all formal parameters of the procedure are bound to the arguments of the procedure call. Then control is transferred to the body of the procedure. This body is executed until its end or a **RETURN** statement is encountered. Then control is transferred back to the point immediately after the point at which the procedure was called. Procedure calls can be nested and so will be the extent of any local variables or parameters procedures created during a call.

The binding of arguments to parameters involves evaluating the arguments. This occurs in an undefined order (e.g. in parallel). Parameter binding may have either call-by-value or call-by-reference semantics for constant parameters and either copy-in-copy-out or call-by-reference semantics for variable parameters. Which of these is used for any single procedure call is left to the compiler. Any program that relies on a certain selection within these possibilities is erroneous.

Design rationale: The appropriateness of one or the other parameter passing mechanism depends on the particular data type to be passed and the actual parallel machine on which the program shall run. Thus, the compiler should have the freedom to choose the most efficient mechanism in each situation.

24 *Procedure Call*[24] ≡

```
{
 ObjectProcedureCall:
    Object '.' ObjectProcedureId '(' ArgumentList ')'.

 ObjectProcedureId:
    LowercaseIdent.
}
```

This macro is defined in definitions 23, 24, and 25.
This macro is invoked in definition 20.

An object procedure call, for which the object is an array or a group of nodes or an input or output interface of a node (referring to a set of connections) is called a *group procedure call*. A group procedure call means that the called object procedure is executed for all objects of the group in an asynchronously parallel fashion (i.e. in any sequential or overlapping order). This language construct introduces a level of object-centered parallelism. Such object-centered parallelism is similar to data parallelism but is more expressive than pure data parallelism, because more than a single assignment or expression can be evaluated in a single parallel statement and additional parallelism can be introduced in the body of an object-centered parallel operation.

Object procedure calls for network procedures are allowed in the central agent and in network procedures and functions. Object procedure calls for individual nodes or for arrays or groups of nodes are allowed in network procedures and network functions. Object procedure calls for individual nodes are allowed in node procedures and functions. Object procedure calls for input or output interfaces of nodes (thus calling a connection type object procedure) is allowed in node procedures and node functions. Object procedure calls for individual connections are allowed in connection procedures and functions.

25 *Procedure Call*[25] ≡

```
{
 MultiObjectProcedureCall:
    ObjectProcedureCall 'AND' ObjectProcedureCall /
```

```
    ObjectProcedureCall 'AND' MultiObjectProcedureCall.
}
```

This macro is defined in definitions 23, 24, and 25.
This macro is invoked in definition 20.

A multiple object procedure call means the execution of the individual object procedure calls in an asynchronously parallel fashion (i.e. in any sequential or overlapping order). This language construct introduces a level of process parallelism. This is the only kind of process parallelism supported in CuPit.

8.5 Reduction statement

Reduction Statement[26] ≡ 26

```
{
 ReductionStmt:
    'REDUCTION' Object ':' ReductionFunctionId 'INTO' Object.

 ReductionFunctionId:
    LowercaseIdent.
}
```

This macro is invoked in definition 20.

Reduction statements are allowed in the central agent, in network procedures and functions and in node procedures and functions. For the meaning of the statement `REDUCTION obj.d:op INTO x` there are three cases:

`obj` can be a network variable. Then the call must be in the central agent and `d` is a data element of the network variable (i.e. not a node or a node group). In this case, the `op` reduction of the `d` elements in all replicates of the network is determined and stored into `x`.

Or `obj` is a group of nodes. Then the call must be in a network procedure and `d` is a data element of the base type of the node group (i.e. the node type). In this case, the `op` reduction of the `d` elements of all nodes of the node group is determined and stored into `x`.

Or `obj` is a connection interface element of a node. In this case, the call must be in a node procedure and `d` must be a data element of the connections attached to the interface. In this case, the `op` reduction of the `d` elements for each node of the node group are determined and stored into `x`.

If the set of objects to perform the reduction on is empty, `x` is not changed. The types of the object to reduce, the object to reduce into, and the reduction function must be the same.

8.6 Winner-takes-all statement

Wta Statement[27] ≡ 27

```
{
 WtaStmt:
    'WTA' Object ':' Elementname '.' WtaFunctionId ':'
    ObjectProcedureId '(' ArgumentList ')'.

 WtaFunctionId:
    LowercaseIdent.
}
```

This macro is invoked in definition 20.

Winner-takes-all statements are allowed in the central agent, in network procedures and functions and in node procedures and functions. For the meaning of the statement `WTA obj:d.op:p(params)` there are three cases:

`obj` can be a network variable. Then the call must be in the central agent and `d` is a data element of the network variable (i.e. not a node or a node group). In this case, the winner of the `d` elements in all replicates of the network with respect to the WTA function `op` is determined and the function `p` is called only for the winning network replicate.

Or `obj` is a group (or array) of nodes. Then the call must be in a network procedure and `d` is a data element of the base type of the node group (i.e. the node type). In this case, the winner of the `d` elements of all nodes of the node group with respect to the WTA function `op` is determined and the function `p` is called only for the winning node in each network replicate.

Or `obj` is a connection interface element of a node. In this case, the call must be in a node procedure and `d` must be a data element of the connections attached to the interface. In this case, the winners of the `d` elements for each node of the node group with respect to the WTA function `op` are determined and the function `p` is called only for the winning connection of each node in each network replicate.

The types of `d` and `op` must be the same. If the set `obj` of objects to pick the winner from is empty, the procedure `p` is not called at all.

8.7 Control flow statements

28 *Return Statement*[28] ≡

```
{
 ReturnStmt:
    'RETURN' /
    'RETURN' Expression.
 }
```

This macro is invoked in definition 20.

The `RETURN` statement is allowed in all kinds of functions and procedures. Its semantics is the immediate termination of the execution of the current procedure or function. In functions (and only in functions) an expression must be given, which must have a type that is compatible to the declared return type of the function. This expression is evaluated and (perhaps after an implicit type conversion to the return type) returned as the result of the function. Since a `RETURN` statement is the only way to return a value in a function, each function must have at least a `RETURN` statement at its end.

In group function or procedure invocations, the `RETURN` statement of course terminates only the calls that execute it, the others continue normal execution.

29 *If Statement*[29] ≡

```
{
 IfStmt:
    'IF' Expression 'THEN' Statements ElsePart 'END' OptIF.

 ElsePart:
    /* nothing */ /
    'ELSE' Statements /
    'ELSIF' Expression 'THEN' Statements ElsePart.

 OptIF:
    /* nothing */ /
    'IF'.
 }
```

This macro is invoked in definition 20.

The semantics of the `IF` statement is the same as in Modula-2.

30 *Loop Statement*[30] ≡

```
{
```

```
LoopStmt:
   OptWhilePart 'REPEAT' Statements OptUntilPart 'END' OptREPEAT /
   'FOR' Object ':=' Expression ForLoopStep Expression
      'REPEAT' Statements OptUntilPart 'END' OptREPEAT.

OptWhilePart:
   /* nothing */ /
   'WHILE' Expression.

OptUntilPart:
   /* nothing */ /
   'UNTIL' Expression.

OptREPEAT:
   /* nothing */ /
   'REPEAT'.

ForLoopStep:
   'UPTO' / 'TO' / 'DOWNTO'.
}
```

This macro is invoked in definition 20.

Loops are available in two forms: the normal loop and the `FOR` loop.

The normal loop can have two boolean conditions, both are optional. This combines the WHILE, UNTIL, and LOOP loop types of Modula-2 and has the intuitive semantics. The `WHILE` test defaults to true and the `UNTIL` test defaults to false. The `WHILE` test is evaluated immediately before each iteration of the loop body, the `UNTIL` test is evaluated immediately after each iteration of the loop body. Whenever a `WHILE` test yields false or an `UNTIL` test yields true, the loop terminates.

Design rationale: You won't need a combined while/until loop very often. But once you need it, it is really nice to have it.

The semantics of the `FOR` loop are be defined by the following transformation pattern: A loop of the form `FOR i := f TO t REPEAT s; UNTIL c END` has the meaning

```
i  := f;                (* initialization *)
t2 := t;                (* limit computation *)
WHILE i <= t2 REPEAT    (* FOR termination test *)
  s;                    (* body *)
  IF c THEN BREAK END;  (* UNTIL termination test *)
  i += 1;               (* count step *)
END
```

where `i` is an existing variable of integral type, `f` and `t` are arbitrary expressions of integral type, `s` is a list of statements, and `c` is a boolean expression. `t2` is an implicitly declared anonymous variable of the same type as `t` that is used for this loop only. The keyword `TO` may be replaced by `UPTO` without change in meaning. It may also be replaced by `DOWNTO`. In this case the "for termination test" is `i >= t2` and the "count step" is `i += -1`. In all three forms, the `UNTIL` test defaults to false, just as for the normal loop.

Break Statement[31] ≡ 31

```
{
 BreakStmt:
    'BREAK'.
 }
```

This macro is invoked in definition 20.

The `BREAK` statement is allowed in loops only. Its semantics is the immediate termination of the innermost textually surrounding loop, just like the `break` statement in C.

8.8 Data allocation statements

32 *Data Allocation Statement*[32] ≡

```
{
 DataAllocationStmt:
    'REPLICATE' Object 'INTO' Expression /
    'EXTEND' Object 'BY' Expression /
    'CONNECT' Object 'TO' Object /
    'DISCONNECT' Object 'FROM' Object.
 }
```

This macro is invoked in definition 20.

These statements allocate or deallocate nodes or connections or create or reunite network replicates.

8.8.1 Connection creation and deletion

The `REPLICATE` statement can in its first form be used in a connection procedure. In `REPLICATE ME INTO n`, the expression must be non-negative integral and gives the number of identical exemplars of this connection that shall exist after the replication statement has been executed. Zero means "delete myself", one means "do nothing". In connection procedures, only `REPLICATE ME INTO 0` and `REPLICATE ME INTO 1` are allowed (*Design rationale:* Only one connection can exist between any two node interfaces at any given time).

The rest of the procedure in which `REPLICATE` was called is not executed, i.e., the `REPLICATE` statement implies a `RETURN`. It is a run time error to call `REPLICATE` for a connection with an operand that is negative or larger than one or to call it while the whole network is replicated.

The `CONNECT` and `DISCONNECT` statements can only be used in network procedures to create or delete connections between two groups of nodes, which have to be given in the order origin–destination. The statement `CONNECT a[2...4].out WITH b[].in1` has the following semantics: `a` and `b` must be node arrays or node groups of the network for which the statement was issued. `out` must be an output interface of the nodes in `a`, `in1` must be an input interface of the nodes in `b`; the types of `in1` and `out` must be identical. The statement creates a connection from each of the nodes 2, 3, and 4 of `a` to each node of `b`. Generally speaking, the objects given in a `CONNECT` or `DISCONNECT` statement must be parallel variable selections (see section 10.3 on page 42) of connection type, where the first one is an output interface and the second an input interface. All newly created connections are initialized using the default initializers given in the respective connection type declaration. Connections that already exist are not created again and are not initialized again.

The `DISCONNECT` statements works in the same way, except that it deletes connections instead of creating them. If `CONNECT` is used to create connections that already exist, an additional exemplar of these connections may or may not be created; such use is non-portable and should be avoided. If `DISCONNECT` is used to delete connections of which multiple exemplars exist, all exemplars will be deleted. It is no error if some or all of the connections that a `DISCONNECT` statement conceptually would delete do not exist. It is a run time error to call `CONNECT` or `DISCONNECT` while the network is replicated. `CONNECT` may produce a run time error if there is not enough memory available on the machine.

8.8.2 Node creation and deletion

The `REPLICATE` statement can in its first form be used in a node procedure. In `REPLICATE ME INTO n`, the expression must be non-negative integral and gives the number of identical exemplars of this node that shall exist after the replication statement has been executed. Zero means "delete myself", one means "do nothing" and larger values mean "create `n`-1 additional exemplars". All incoming and outgoing connections of the node are cloned for each new exemplar when `REPLICATE` is called with a value of 2 or higher. The new nodes are inserted in the index range at the point of the old node (i.e. the replicates of a node will be in a contiguous subrange of the new index range). The new indices are computed in a way that maintains the order of the indices of the nodes (although not the indices itself). Example: In a node group with four nodes 1, 2, 3, 4, after a replicate statement where the nodes request 3, 1, 0, 1 replicates, respectively, the new indices 1, 2, 3, 4, 5 will be given to the nodes stemming from the nodes with old indices 1, 1, 1, 2, 4, respectively. The statement can produce a run time error if it creates so many new nodes that the machine runs out of memory, if it is called for nodes that are not part of a `GROUP` but part of a node `ARRAY` instead, and if it is called while the network is replicated.

`REPLICATE` implies `RETURN`, i.e., the procedure that calls it terminates after the replication has been performed. *Open question: This is a bit ugly. But what is the semantics otherwise? And how would you implement it?*

The `EXTEND` statement can only be used in network procedures for nodes that belong to a node `GROUP`. `EXTEND g BY n` means that the group of nodes `g` shall be extended by `n` new nodes (or reduced by `-n` nodes if `n` is negative). The nodes are added or removed at the upper end of the group's current index range. The new nodes, if any, are initialized using the default initializers as given in the type declaration of the node type, if any. The new nodes do not have any connections initially. It is a run time error if the size that the group `g` would have after the `EXTEND` is negative, if `EXTEND` is called while the network is replicated, or if there is not enough memory on the machine.

8.8.3 Network replication

The network replication statement is allowed in the central agent only. The object must be a network variable. The expression must have integral or integer interval type.

Design rationale: At the beginning of the existence of a network variable, the corresponding object exists as a single exemplar (as one would usually expect for any variable of any type). Since many Neural Algorithms allow input example parallelism, i.e., the simultaneous independent processing of several input examples, CuPit allows network objects to be *replicated*. This is what the network replication statement is for.

`REPLICATE nw INTO 3`, for example, tells CuPit to create 3 exemplars of the network object designated by the variable `nw`. The exemplars are identical copies of the original object. The input assignment and output assignment statements, though, allow to feed different data into and read different data from each of the exemplars. `REPLICATE nw INTO 3...20`, tells CuPit to create any number of exemplars of the network object it would like to, provided it is in the range 3 to 20. The compiler chooses the number of replicates that it thinks will make the program run fastest.

Design rationale: The compiler may have a lot more knowledge about available memory and the cost of replicating, reuniting (merging), and operating on several replicates in parallel than the programmer has. It should thus be given some freedom to optimize the parameter "number of network replicates". A compiler may for example choose to prefer network replication with numbers of replicates that are powers of two, because this is the most efficient on the particular target machine.

While a network is replicated, all network procedure calls are automatically executed by all exemplars of the network. For network functions the behavior is different, depending on where they are being called from: If a network function is called from the central agent or from another network function that has been called from the central agent, the function is executed and the results are returned for the first exemplar of the network only. If it is called from a network procedure or from another network function that has been called from a network procedure, execution occurs on all exemplars of the network and a value is returned for all exemplars as well.

`REPLICATE nw INTO 1` reunites the replicated exemplars to a single object again, using the `MERGE` procedures as defined in the network type and the relevant node and connection types. The two states 'replicated' and 'non-replicated' have an important difference: While a network is replicated, no `CONNECT, DISCONNECT, REPLICATE,` or `EXTEND` commands must be issued for its parts. This restriction is necessary because it is not clear how replicates with differing topology could be merged. The advantage of the restriction is that it may allow the compiler to work with a more efficient data distribution in replicated state. Even if a program uses only one replicate all the time, it can switch between "topology changes allowed but data distribution maybe less efficient" and "topology changes forbidden but data distribution is most efficient" by using `REPLICATE nw INTO 1...1` for the latter.

The number of exemplars minus one that currently exist can be inquired for any network variable using the `MAXINDEX` operation. It is a run-time error, to request a number of replicates that is not strictly positive or to request network replication while the network is already replicated.

8.9 Merge statement

33 *Merge Statement*[33] $\equiv$

```
{
 MergeStmt:
    'MERGE' Object.
}
```

This macro is invoked in definition 20.

The statement `MERGE nw` applies the respective `MERGE` procedures to all parts of all replicates of the network `nw`, thus collecting the data from all the replicates in the first replicate, and then redistributes this data from the first replicate to all other replicates again. After a `MERGE`, the values of all corresponding data elements that are merged by the merge procedures of the respective data types are identical in the different network replicates. It is undefined whether the data elements not modified by the individual `MERGE` procedures retain their previous values in all replicates or are all changed to the values of the corresponding data elements of the first replicate. The `MERGE` statement can only be called from the central agent.

Design rationale: It is often useful to reunite the data in network replicates without actually destroying the replicated network, because the next thing the program does is to create replicates again, anyway. This is the case when the purpose of reuniting the replicates is not a change in network topology but only the collection of data from the replicates.

9 Expressions

9.1 Overview

Most of the expression syntax and semantics of CuPit is well-known from common procedural languages: Mentioning an object uses it as a value, a function can be called with arguments and returns a value, operators are used to combine values generating new values, all values have a type, there are restrictions on type compatibility for the application of operators, and values of some types can explicitly be converted into values of other types. There are, though, a few special expressions, which are concerned with handling dynamic data structures and accessing object elements. The concrete operators that are available can be seen in table 1.

9.2 Type compatibility and type conversion

For most binary operations (including assignment and parameter passing), the two operands must be *compatible*. In the current version of CuPit, two types A, B are compatible only if they are the same; the exception to this rule is automatic promotion from smaller to larger integer types and integer interval types according to the following rules: Two integer types A and B are compatible if and only if either

1. they are the same or
2. A is smaller than B and A is `not` the type of the left-hand object in an assignment or the formal parameter in an argument passing, or
3. B is smaller than A and B is `not` the type of the left-hand object in an assignment or the formal parameter in an argument passing.

Prio	Appearance	Purpose
1	?:	ternary if-then-else expression
2	OR	Boolean or
2	XOR	Boolean exclusive or
3	AND	Boolean and
4	= <> < >	Equality, inequality, less than, greater than
4	<= >=	Less than or equal, greater than or equal
4	IN	Interval hit
5	BITOR	Bitwise or
5	BITXOR	Bitwise exclusive or
5	...	Interval construction
6	BITAND	Bitwise and
7	LSHIFT	Leftshift
7	RSHIFT	Rightshift
8	+ -	Addition
9	* / %	Multiplication, Division, Modulo
10	**	Exponentiation
11	NOT	Unary boolean not
11	BITNOT	Unary bitwise not
11	-	Unary arithmetic negation
11	MIN	Access minimum of interval
11	MAX	Access maximum of interval
11	RANDOM	Random number generation
11	-	Unary arithmetic negation
11	Type	Explicit type conversion or construction
12	[]	Array/group subscription, parallel variable creation
13	.	Record element selection
14	()	Grouping

Table 1: Operators in CuPit

In the latter two cases, the smaller operand is converted into the type of the larger one. Formal parameters can not be converted, nor can objects that are passed as arguments to a `VAR` or `IO` formal parameter. Integer denoters have smallest integer type that can represent their value. Analogous rules apply to integer intervals.

For explicit type conversion, see page 40. The set of explicit type conversions that are available can be described as follows. There are type constructors that generate an object of a certain type T from objects of the component types of X: For each record type there is a conversion from a complete set of record elements to the record type, e.g. an object of `TYPE Rec IS RECORD REAL a; INT b; BOOL c; INT d; END` can be constructed by `Rec(3.0,7,false,0)`. The order of the arguments for the conversion is the order in which the elements of the record were defined. Type constructors for array or group types do not exist.

9.3 Operators

34 *Expression*[34] ≡
```
{
 Expression:
    E1.

 ExprList:
    Expression /
    ExprList ',' Expression.
}
```
This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

All operators can be used in a constant expression. The only requirement is that the values of all operands must be available at compile time. The compiler performs as much constant folding as possible with real, integer, and boolean values in order to produce constant expressions where necessary. The compiler may, but need not, fold constants in other contexts, too. Note that this may change the semantics of a program, if the compilation machine's arithmetic is not exactly equivalent to that of the target machine.

35 *Expression*[35] ≡
```
{
 E1:
    E2 '?' E2 ':' E2 /
    E2.
}
```
This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

This is the useful if-then-else operator known from C. Note that it is non-associative in CuPit: In order to nest it, parentheses must be used. The first expression must have boolean type, the second and third must have compatible types.

`A ? B : C` has the following semantics: First, `A` is evaluated and must yield a `Bool`. Then, if `A` is true, `B` is evaluated and returned, otherwise `C` is evaluated and returned. The types of `B` and `C` must be compatible; implicit type conversion is performed on them as if they were an operand pair.

36 *Expression*[36] ≡
```
{
 E2:
    E2 OrOp E3 /
    E3.

 OrOp:
```

```
    'OR' / 'XOR'.

  E3:
    E3 AndOp E4 /
    E4.

  AndOp:
    'AND'.
  }
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

These are the usual logical operators: both of their operands must have type `Bool`, the result has type `Bool`, too. `a OR b` is true iff either `a` or `b` or both are true. `a XOR b` is true iff either `a` or `b` but not both are true. `a AND b` is true iff both, `a` and `b`, are true. The operands are evaluated in undefined order. *Open question: Do we need this freedom ? Or would it be better to define left-to-right shortcut evaluation ?*

Expression[37] ≡ 37

```
 {
  E4:
    E5 CompareOp E5.

  CompareOp:
    '='  /  '<>'  /  '<'  /  '>'  /  '<='  /  '>='.
  }
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

These are the usual comparison operators: both of their operands must be numbers or enumerations; their types must be compatible. The result has type `Bool`. The result for the comparison of `SYMBOLIC` values is well-defined only for the `'='` and `'<>'` test. The other tests yield a result without any special meaning for these types, but this result is constant within the same run of the program. The operands are evaluated in undefined order.

Expression[38] ≡ 38

```
 {
  E4:
    E5 InOp E5 /
    E5.

  InOp:
    'IN'.
  }
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

This is the interval test operator. The left operand must have a number type, the right operand must have the corresponding interval type. The `IN` operator returns true if the value of the left operand lies in the interval and false otherwise.

Expression[39] ≡ 39

```
 {
  E5:
    E5 BitorOp E6.

  BitorOp:
    'BITOR' / 'BITXOR'.
  }
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

These are logical operators that operate bitwise. Both of their operands must be integral numbers; their types must be compatible. The operation `a BITOR b` means that for every bit position in the internal representation of `a` and `b` (after the type conversion required by the compatibility has been performed) a logical OR operation is performed, just as the `OR` operator does. A zero bit corresponds to false and a one bit corresponds to true. `BITXOR` is defined analogously. Since the internal representation of integral numbers is not defined in CuPit, the result of these operators is generally machine-dependent. The operands are evaluated in undefined order.

40 *Expression*[40] $\equiv$

```
{
 E5:
    E6 IntervalOp E6 /
    E6.

 IntervalOp:
    '...'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

This is the interval construction operator: both operands must be numbers and must have compatible types. The result of `a...b` is a `Realerval` if `a` and `b` have type `Real` and an `Interval` if `a` and `b` have types compatible to `Int`. Objects of type `Interval1` and `Interval2` can only be generated by explicit type conversion.

The interval is empty, if `a` $>$ `b`, otherwise it contains all numbers x of type `Int` or `Real`, respectively, for which `a`$\leq x \leq$`b`. `a` and `b` are evaluated in undefined order.

41 *Expression*[41] $\equiv$

```
{
 E6:
    E6 BitandOp E7 /
    E7.

 BitandOp:
    'BITAND'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

This is the bitwise logical AND operator. Works analogous to `BITOR`, but performs a bitwise `AND` operation.

42 *Expression*[42] $\equiv$

```
{
 E7:
    E7 ShiftOp E8 /
    E8.

 ShiftOp:
    'LSHIFT' / 'RSHIFT'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

These are shift operators working on the bit representation of the left operand. Both operands must be of integral type. The result for negative values of `a` or `b` is machine-dependent; otherwise `a LSHIFT b` (where `a` has type `A`) is equivalent to `A(a*2**b)` and `a RSHIFT b` (where `a` has type `A`) is equivalent to `A(a/2**b)`. The operands are evaluated in undefined order.

Expression[43] ≡ 43

```
{
 E8:
    E8 AddOp E9 /
    E9.

 AddOp:
    '+' / '-'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

Addition and subtraction of numbers. Both operands must be of compatible type. The exact semantics of these operations is machine-dependent (but will be the same on almost all machines). The operands are evaluated in undefined order.

Expression[44] ≡ 44

```
{
 E9:
    E9 MulOp E10 /
    E10.

 MulOp:
    '*' / '/' / '%'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

Multiplication, division, and modulo operation on numbers. For multiplication and division, both operands must have compatible type. The exact semantics of these operations is machine-dependent (but will be the same on almost all machines, except perhaps for division and modulo by negative integers). For modulo, the right operand must have integral type. `a % b` where `a` is integral is defined as `a-b*(a/b)` for positive `a` and `b` and is machine-dependent if either is negative. `a % b` where `b` has type `Real` is defined as `a-b*R(Int((a/b)))`. The operands are evaluated exactly once, in undefined order.

Expression[45] ≡ 45

```
{
 E10:
    E11 ExponOp E10 /
    E11.

 ExponOp:
    '**'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

The exponentiation `a**b` is defined

1. for integral `a` and non-negative integral `b` with result type `Int`.
2. for real `a` and integral `b` with result type `Real`.
3. for non-negative real `a` and arbitrary real `b` with result type `Real`.

In all cases, the meaning is a^b, where 0^0 equals 1. The behavior upon overflow is machine-dependent. The compiler may provide run-time checking. The operands are evaluated in undefined order.

Expression[46] ≡ 46

```
{
 E11:
    UnaryOp E12 /
    TypeId '(' ExprList ')' /
    'MAXINDEX' '(' Object ')' /
    E12.

 UnaryOp:
    'NOT' / 'BITNOT' / '-' / 'MIN' / 'MAX' / 'RANDOM'.
}
```

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

These are unary operators. All unary operators have the same precedence.

`NOT a` is defined iff `a` has type `Bool`; it returns false, if `a` is true and true if `a` is false.

`BITNOT a` is defined iff `a` has an integer type. The internal representation of `a` is returned complemented bitwise. The result has the same type as `a`.

`-a` is defined iff `a` has a number type. The result is the same as `(0-a)`.

`MIN(a)` and `MAX(a)` are defined iff `a` has an interval type. The result is the minimum or maximum, respectively, of the interval.

`RANDOM a` is defined for real or integer intervals `a`. It returns a pseudorandom number in the given interval, with even distribution. This operator can usually not be evaluated as a constant expression, i.e., each time `RANDOM a` is executed at run time, it may return a different value, even if `a` is not changing. The exception to this rule occurs when it is possible to guarantee that the expression will be evaluated only once; this is always the case for the initialization of global variables.

A typename `X` can be applied to a parenthesized expression like a unary operator in order to specify a type conversion into type `X`. All the usual conversions between the types `Real, Int, Int1, Int2` are available; their exact semantics is machine-dependent.

For structure types, it is possible to list all data elements of an object of that type separated by commas and enclosed in parentheses in order to use the typename as a constructor for values of that type. The elements must appear in the order in which they are defined in the type definition.

`MAXINDEX(a)` returns the highest currently available index to the object `a` as an `Int`. `a` must be a connection interface, group, array, or network. For connection interfaces, the number of connections at the interface minus one is returned. For networks the meaning is the number of currently existing replicates minus one.

47 *Expression*[47] ≡

```
{
 E12:
    '(' Expression ')' /
    Object /
    Denoter /
    FunctionCall /
    ObjectFunctionCall.
```

Data Object Access[50]
Denoter[48]
Function Call[49]
}

This macro is defined in definitions 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, and 47.
This macro is invoked in definition 55.

48 *Denoter*[48] ≡

```
{
 Denoter:
    IntegerDenoter /
    RealDenoter /
    StringDenoter.
 }
```

This macro is invoked in definition 47.

9.4 Function call

Function Call[49] ≡ 49

```
{
 FunctionCall:
    FunctionId '(' ArgumentList ')'.

 FunctionId:
    LowercaseIdent.

 ObjectFunctionCall:
    Object '.' ObjectFunctionId '(' ArgumentList ')'.

 ObjectFunctionId:
    LowercaseIdent.
 }
```

This macro is invoked in definition 47.

Function calls look exactly like procedure calls. The difference is that functions return a value. This value can usually be used just like any other value of the same type. Function calls that involve parallelism, however, are allowed only if they do not change the amount of parallelism: Object function calls to network functions from within network functions or network procedures and object function calls to node functions from within node functions or node procedures and object function calls to connection functions from within connection functions or connection procedures work just like other normal function calls; object function calls to node functions from network functions or network procedures and object function calls to connection functions from node functions or node procedures are not allowed. Object function calls to network functions from the central agent are an exception: They return the result from the first network replicate. *Design rationale:* We could allow function calls into a higher level of parallelism instead of the data element in a reduction statement. I didn't do it in order to keep the semantics and implementation of the reduction statement simple.

10 Referring to data objects

Data objects are referred to either by identifiers, by record element selection, by subscription, by connection addressing, or by special keywords.

10.1 ME, YOU, INDEX, and explicit variables

Data Object Access[50] ≡ 50

```
{
 Object:
    Objectname /
    'ME' /
```

```
    'YOU' /
    'INDEX'.

 Objectname:
    LowercaseIdent.
}
```

This macro is defined in definitions 50, 51, 52, and 53.
This macro is invoked in definition 47.

An identifier used as an object refers to an object by name. We call this an *explicit variable* (using the term "variable" also for `CONST` and `IO` objects, meaning an object occupying some data storage — as opposed to, say, a denoter).

The special object `ME` can only be used in object subroutines, object merge procedures, and winner-takes-all and reduction functions. It denotes the object for which the procedure or function was called. `ME` is called an *implicit variable*. In reduction and winner-takes-all functions, `ME` is `CONST`, otherwise it is `VAR`.

The special object `YOU` can only be used in object merge procedures and winner-takes-all and reduction functions. It denotes the second object for which the procedure or function was called and is also an implicit variable. For merge procedures, the result of the merging has to be constructed in `ME` (i.e. `YOU` has to be merged into `ME` — not the other way round). For reduction functions, the value constructed from `ME` and `YOU` is returned as the function result (i.e. here `ME` and `YOU` have equal rights). `YOU` is always `CONST`.

The special object `INDEX` is an implicit `Int CONST` and can only be used in object procedures and object functions. When read in a network procedure, it returns the replicate index of the network replicate it is read from. When read in a node subroutine, its value is the current index number of the object for which the subroutine has been called in the array or group it belongs to. `INDEX` is undefined in connection subroutines.

Note that for the very same node object, the value of `INDEX` may change from call to call if sister objects are created or deleted with the `REPLICATE` statement.

10.2 Selection

51 *Data Object Access*[51] ≡

```
{
 Object:
    Object '.' Elementname.

 Elementname:
    LowercaseIdent.
}
```

This macro is defined in definitions 50, 51, 52, and 53.
This macro is invoked in definition 47.

Selections pick an element of a structure type object (node, connection, network, or record) by name. The selected element can be a data element (of a network, node, connection, or record), a node group or node array (of a network), or an interface element (of a node).

10.3 Subscription and parallel variables

52 *Data Object Access*[52] ≡

```
{
 Object:
    Object '[' Expression ']' /
    Object '[' ']'.
```

```
    }
```
This macro is defined in definitions 50, 51, 52, and 53.
This macro is invoked in definition 47.

Subscriptions pick one or several elements of an array or group by position. To pick a single element, the expression must have integral type. If the value of the expression is n, e.g. `ME.nodes[n]`, the subscription refers to the element of the array or group that has index n. The first object of an array or group has index 0. To pick several elements at once, the expression must have `Interval` type, e.g. `ME.nodes[3...5]`. The object that is referred to by such a subscription is called a *slice* of the group or array and consists of all elements whose index is contained in the interval. There is a special slice, called the *all-slice* containing all objects of an array or group that is denoted by just using selection brackets without any expression at all, e.g. `ME.nodes[]`. Slice subscriptions that contain indices of objects that are not existing in the sliced object are automatically clipped to only the existing objects without producing an error.

As we have seen in the description of the object procedure calls, all parallelism in CuPit is created by operating on a multitude of objects. To describe this, the notion of a *parallel variable* is used: a parallel variable is a multiple object that can induce parallelism; parallel variables are either multiple replicates of a network, multiple nodes of a node array or group, or multiple connections attached to a connection interface of a node.

Slice subscription is used to create parallel variables. In order to do this, connection interfaces and explicit network variables are implicitly treated as groups . Given a network variable called `net`, a node group (or array) called `nodes` and a connection interface called `cons`, we find the following cases: In a global subroutine, `net[]` and `net[1...3]` are parallel variables while `net` and `net[3]` are ordinary variables. In a network subroutine `ME.nodes[]` and `ME.nodes[3...5]` are parallel variables while `ME.nodes` and `ME.nodes[3]` are ordinary variables. In a node subroutine, `ME.con[]` is a parallel variable while `ME.con` is an ordinary variable; actual subscription is not allowed for connection interfaces.

Parallel variables can be used for calls to object procedures (but not object functions, since that would return a multitude of values). Further subscription is not allowed on parallel variables. Further selection from a parallel variable creates a *parallel variable selection*. Such an object can be used only in `REDUCTION` and `WTA` statements and in `CONNECT` and `DISCONNECT` statements. E.g. given a `Real` data element called `r` in `cons`, in `REDUCTION ME.cons[].r:sum INTO x` the term `ME.cons[]` denotes a parallel variable of connection type and `ME.cons[].r` is a parallel variable selection of `Real` type.

10.4 Connection addressing

Data Object Access[53] ≡ 53
```
  {
   Object:
      '{' Object '-->' Object '}'.
    }
```
This macro is defined in definitions 50, 51, 52, and 53.
This macro is invoked in definition 47.

The connection addressing syntax uses the right arrow to address a connection by giving the node output interface from which it originates and the node input interface at which it ends, e.g. `{net.nd[1].out-->net.hid[1].in}`. Such objects can be used to create and initialize connections at the beginning of the program run. They may appear on the left hand side of assignments only, cannot be further selected, and have the side effect to create the connection described by the object pair. Both node interfaces must belong to nodes in the same network. The construct can only be used in the central agent and only while the number of network replicates is 1.

11 The central agent

All global (i.e. non-object) procedures and functions of a CuPit program either belong to the *central agent* or are called *free*. The *central agent* of a CuPit program consists of the global procedure with the name `program` plus a number of other subroutines according to the rules given below.

Design rationale: The significance of the central agent is that certain operations are allowed only there. The idea behind the central agent is that it is the (sequential) control program from which the (possibly parallel) network operations are called. All parallelism occurs outside the central agent hidden in object procedures.

A function or procedure is free if and only if

1. it does not mention a `NETWORK` variable explicitly and
2. it does not call any global procedure or function that is not free

All subroutines that are not free are part of the central agent.
All subroutines that are part of the central agent are not free.

The global procedure `program` must exist and is always part of the central agent (unless the program does not use a `NETWORK` variable at all); the `program` procedure is implicitly called when a CuPit program is invoked. Object subroutines are never part of the central agent. Note that since the CuPit compiler cannot check external procedures they are always assumed to be free. It is not allowed to call subroutines that belong to the central agent from an object subroutine. Object-subroutines may, however, call free global subroutines.

12 Overall program structure

A CuPit program is simply a sequence of type definitions, data object definitions, and procedure or function definitions. Any object must be defined before it can be used.

54 *Cupit Program*[54] ≡

```
{
 Root:
    CupitProgram.

 CupitProgram:
    CupitParts.

 CupitParts:
    /* nothing */ /
    CupitParts CupitPart ';'.

 CupitPart:
    TypeDef /
    DataObjectDef /
    ProcedureDef /
    FunctionDef /
    ReductionFunctionDef /
    WtaFunctionDef.
 }
```

This macro is invoked in definition 55.

All these definitions are now put into the Eli [GHL+92] grammar specification file `grammar.con`:

55 **grammar.con**[55] ≡

{
Cupit Program[54]
Type Definition[1]
Subroutine Definition[13]
Data Object Definition[12]

Statement[20]
Expression[34]
}
This macro is attached to an output file.

13 Basic syntactic elements

All keywords and operators in a CuPit program must appear exactly as shown in the grammar. The syntactic structure of identifiers, denoters (value literals), and comments will be described in this section.

These are the contents of the scanner definition for CuPit:

scanner.gla[56] ≡ 56
{
Lowercase Identifier[57]
Uppercase Identifier[58]
Integer Denoter[59]
Real Denoter[60]
String Denoter[61]
Wrong Keywords[62]
Comment[64]
}
This macro is attached to an output file.

13.1 Identifier

Identifiers appear in two forms: Starting with an uppercase letter (for type names) or starting with a lowercase letter (for everything else).

Lowercase Identifier[57] ≡ 57
{
```
LowercaseIdent:  $[a-z][a-zA-Z0-9]*   [mkidn]
```
}
This macro is invoked in definition 56.

A `LowercaseIdent` is a sequence of letters and digits that starts with a lowercase letter.

Uppercase Identifier[58] ≡ 58
{
```
UppercaseIdent:  $([A-Z][a-zA-Z0-9]*[a-z0-9][a-zA-Z0-9]*)|[A-Z]   [mkidn]
```
}
This macro is invoked in definition 56.

An `UppercaseIdent` is either a single uppercase letter or a sequence of letters and digits that starts with an uppercase letter and contains at least one lowercase letter or digit. This has the consequence that for instance `T`, `T1` and `TreeIT2` are `UppercaseIdent`s while `TREE` is not.

13.2 Denoter

There are denoters for integer, real, and string values.

Integer Denoter[59] ≡ 59
{
```
IntegerDenoter:  $([0-9]+|0[xX][0-9a-fA-F]*)   [c_mkint]
```

```
}
```
This macro is invoked in definition 56.

Integer denoters are defined exactly as in the C programming language, except that the `L` and `U` suffixes are not supported in CuPit.

60 *Real Denoter*[60] ≡
```
{
RealDenoter:  $([0-9]+\.[0-9]+)([eE][\+\-]?[0-9]+)?   [mkstr]
}
```
This macro is invoked in definition 56.

Real denoters are similar to floating point denoters in the C programming language. However, there must always be a decimal point that is surrounded by digits in a CuPit floating point denoter.

61 *String Denoter*[61] ≡
```
{
StringDenoter:  $\"  (auxCString)   [c_mkstr]
}
```
This macro is invoked in definition 56.

String denoters are defined exactly as string literals in the C programming language.

13.3 Keywords and Comments

62 *Wrong Keywords*[62] ≡
```
{
$[A-Z][A-Z]+   [ComplainKeyword]
}
```
This macro is invoked in definition 56.

Eli extracts the keywords from the parser grammar and automatically constructs the scanner in a way to recognize them. However, if you misspell a keyword (or use a nonexisting one) you get one syntax error per character in your wrong keyword after the point where the wrong keyword looks different from any existing one. This is awful. Therefore, we introduce a scanner rule that catches any token that looks like a keyword (but is not a true keyword — those always take precedence) and produces an error message that says "I have never heard of a keyword like that and do not like it, too". Here is the procedure that produces this message:

63 **scanerr.c**[63] ≡
```
{
#include "err.h"

void ComplainKeyword (char *start, int lgth, int *extCodePtr, char *intrPtr)
{
  message (ERROR, "Huh ?  What's that ??  An unknown keyword!", 0, &curpos);
}
}
```
This macro is attached to an output file.

64 *Comment*[64] ≡
```
{
$\(\*  (auxM3Comment)
}
```
This macro is invoked in definition 56.

Comments are defined exactly as in Modula-2 or in Modula-3, i.e. comments begin with `(*`, end with `*)`, and can be nested. `auxM3comment` is a so-called "canned description" in Eli; so to say a miniature re-usable module.

14 C preprocessor support

It is convenient if CuPit programs can be run through the C preprocessor before compilation. This allows to store several variants of a program in one file by using the conditional compilation feature of the preprocessor (`#ifdef` etc.).

To allow this, we need support in the compiler to understand the `#line` directives added to its output by the preprocessor. Without such support, we would have to generate preprocessor output without the directives and the CuPit compiler could not produce proper file names and line numbers in its error messages.

For this support, the scanner description must be augmented. When the scanner sees the hash symbol, which starts any preprocessor directive, it calls the auxiliary scanner `auxLinedirective` that reads the directive and adjusts the scanners internal `curpos` accordingly.

cpp.gla[65] ≡ 65

{
```
$# (auxLinedirective)
```
}

This macro is attached to an output file.

The next subsections will contain the implementation of a modified error module and the auxiliary scanner.

14.1 Reimplementation of the `err` module

For the implementation of the auxiliary scanner, we must be able to store the virtual file name given in the `#line` directive in the `curpos` variable. This filename should then be used in error messages referring to this position. Therefore, we need a different definition of the `POSITION` type and an implementation of the `message` function that is changed accordingly. Both are implemented by the following hacked version of the Eli 3.5.1 `err` module. Note that the changes in the data structure definition may affect other modules of Eli that use knowledge about the `POSITION` type.

In `err.h`, the only change compared to the original version is in the definition of the type `POSITION` (plus a new macro `NameOf` to access the filename component). The changes are marked by the comment `LP` in the source code.

err.h[66] ≡ 66

{
```
#ifndef ERR_H
#define ERR_H

/* $Id: err.h,v 1.23 1993/10/14 00:58:46 waite Exp $ */
/* Copyright 1989, The Regents of the University of Colorado
 * Permission is granted to use any portion of this file for any purpose,
 * commercial or otherwise, provided that this notice remains unchanged.
 */

#if defined(__cplusplus) || defined(__STDC__)
#include <stdio.h>
#endif

/* Error report classification */

#define NOTE    0  /* Nonstandard construct */
#define COMMENT 0  /* Obsolete */
#define WARNING 1  /* Repairable error */
```

```
#define ERROR   2  /* Unrepairable error */
#define FATAL   2  /* Obsolete */
#define DEADLY  3  /* Error that makes continuation impossible */


/* Types exported by the Error Module */

typedef struct {  /* Source listing coordinates */
        int line;          /* Line number */
        int col;           /* Character position */
        char *fn;  /*LP*/  /* file name (real or virtual) */
} POSITION;
#define NoPosition      ((POSITION *)0)
#define LineOf(pos)     ((pos).line)
#define ColOf(pos)      ((pos).col)
#define NameOf(pos)     ((pos).fn)  /*LP*/

        /* Variables exported by the Error Module */

extern int ErrorCount[];
extern int LineNum;     /* Index of the current line in the total source text */
extern POSITION NoCoord;  /* The NULL coordinate */
extern POSITION curpos; /* Position variable for general use */
#ifdef MONITOR
extern POSITION endpos; /* Ending position */
#endif

        /* Routines exported by the Error Module */

#if defined(__cplusplus) || defined(__STDC__)
extern void ErrorInit(int ImmOut, int AGout, int ErrLimit);
/* Initialize the error module
 *    On entry-
 *       ImmOut=1 if immediate error output required
 *       AGout=1 to print AG line number on error reports
 *       ErrLimit=1 to limit the number of errors reported
 ***/


extern void message(int severity, char *Msgtext, int grammar, POSITION *source);
/* Report an error
 *    On entry-
 *      severity=error severity
 *      Msgtext=message text
 *      grammar=identification of the test that failed
 *      source=source coordinates at which the error was detected
 ***/


extern void lisedit(char *name, FILE *stream, int cutoff, int erronly);
/* Output the listing with embedded error messages
 *    On entry-
 *       name is the source file name
 *       stream specifies the listing file
 *       cutoff=lowest severity level that will be listed
```

```
 *    If erronly != 0 then on exit-
 *       Source file lines containing errors have been added to file stream
 *          with error messages attached
 *    Else on exit-
 *       All source file lines have been added to file stream
 *          with error messages attached to those containing errors
 ***/
#else
extern ErrorInit();
extern void message();
extern void lisedit();
#endif

#endif
}
```

This macro is attached to an output file.

In `err.c`, the only changes compared to the original version are in the definition of the function `message`, the initialization of the `NoCoord` variable, and the computation of the error limit. The latter has nothing to do with the other changes. All changes are marked by the comment `LP`.

err.c[67] ≡ 67

```
{
static char RCSid[] = "$Id: err.c,v 1.32 1993/09/29 21:48:51 kadhim Exp $";
/* Copyright 1989, The Regents of the University of Colorado
 * Permission is granted to use any portion of this file for any purpose,
 * commercial or otherwise, provided that this notice remains unchanged.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "err.h"
#include "source.h"


        /* Variables exported by the Error Module */

int ErrorCount[] = {            /* Counts at each severity level */
  0, 0, 0, 0
  };
int LineNum = 1;                /* Index of the current line
                                   in the total source text */

POSITION NoCoord = { 0, 0, 0 }; /*LP*/   /* The NULL coordinate */

POSITION curpos;                /* Position variable for general use */
#ifdef MONITOR
POSITION endpos;                /* Ending position */
#endif

static char *key[] = {"NOTE", "WARNING", "ERROR", "DEADLY"};

struct msg {
  int severity;
  POSITION loc;
```

```
  int grammar;
  char *Msgtext;
  struct msg *forward, *back;
};


static struct msg reports = {   /* Error report list */
  DEADLY, 0, 0, 0/*LP*/, 0, "", &reports, &reports};

static struct msg emergency;    /* In case malloc fails */

static int ImmediateOutput = 1; /* 1 if immediate error output required */
static int GrammarLine = 1;     /* 1 to print AG line number */
static int ErrorLimit = 1;      /* 1 to abort after too many errors */

#if defined(__cplusplus) || defined(__STDC__)
void
ErrorInit(int ImmOut, int AGout, int ErrLimit)
#else
ErrorInit(ImmOut, AGout, ErrLimit)
int ImmOut, AGout, ErrLimit;
#endif
/* Initialize the error module
 *    On entry-
 *       ImmOut=1 if immediate error output required
 *       AGout=1 to print AG line number on error reports
 *       ErrLimit=1 to limit the number of errors reported
 ***/
{
#ifdef MONITOR
  generate_enter ("message");
#endif

  ImmediateOutput = ImmOut;
  GrammarLine = AGout;
  ErrorLimit = ErrLimit;

  reports.severity = DEADLY;
  reports.loc.line = reports.loc.col = 0;
  reports.grammar = 0; reports.Msgtext = "";
  reports.forward = reports.back = &reports;

#ifdef MONITOR
  generate_leave ("message");
#endif
}

#if defined(__cplusplus) || defined(__STDC__)
int
earlier(POSITION *p, POSITION *q)
#else
int
earlier(p,q)
POSITION *p, *q;
#endif
```

```
/* Check relative position
 *    On exit-
 *       earlier != 0 if p defines a position in the source Msgtext that
 *          preceeds the position defined by q
 ***/
{
  if (p->line != q->line) return(p->line < q->line);
  return(p->col < q->col);
}


/***/
#if defined(__cplusplus) || defined(__STDC__)
void
lisedit(char *name, FILE *stream, int cutoff, int erronly)
#else
void
lisedit(name, stream, cutoff, erronly)
char *name; FILE *stream; int cutoff, erronly;
#endif
/* Output the listing with embedded error messages
 *    On entry-
 *       name is the source file name
 *       stream specifies the listing file
 *       cutoff=lowest severity level that will be listed
 *    If erronly != 0 then on exit-
 *       Source file lines containing errors have been added to file stream
 *          with error messages attached
 *    Else on exit-
 *       All source file lines have been added to file stream
 *          with error messages attached to those containing errors
 ***/
{
  register char *p;
  int fd;
  struct msg *r;

#ifdef MONITOR
  generate_enter ("message");
#endif

  if (name == NULL || *name == '\0') {
    (void)fprintf(stderr, "lisedit: Null source file name\n");
#ifdef MONITOR
    generate_leave ("message");
#endif
    exit(1);
  }
  if ((fd = open(name,0)) < 0) {
    perror(name);
#ifdef MONITOR
    generate_leave ("message");
#endif
    exit(1);
  }
```

```
  initBuf(name, fd);
  p = TEXTSTART; LineNum = 1;
  r = reports.forward;
  while (r != &reports && r->loc.line == 0) {
    if (r->severity >= cutoff){
      (void)fprintf(stream, "*** %s: %s\n", key[r->severity], r->Msgtext);
    }
    r = r->forward;
  }
  while (r != &reports || (!erronly && *p != 0)) {
    if (r != &reports && LineNum > r->loc.line) {
                        /* Output reports for the last line printed */
      char buf[BUFSIZ];
      int l, s;

      if (r->severity >= cutoff) {
        (void)sprintf(buf, "*** %s: %s", key[r->severity], r->Msgtext);
        l = strlen(buf);
        s = r->loc.col - 1 + (erronly?8:0);
        if (l > s) {
          while (s--) (void)putc(' ', stream);
          (void)fprintf(stream, "^\n%s\n", buf);
        } else {
          (void)fprintf(stream, "%s", buf);
          while (l < (s--)) (void)putc('-', stream);
          (void)fprintf(stream, "^\n");
        }
      }
      r = r->forward;
    } else { /* Print up through the next line with a report */
      register char c;
      char *StartLine = p;

      while ((c = *p++) && c != '\n') ;
      if (c == '\n') {
        if (!erronly || LineNum == r->loc.line) {
          if (erronly) (void)fprintf(stream, "%6d |", LineNum);
          (void)fwrite(StartLine, p-StartLine, 1, stream);
        }
        if (*p == 0) { refillBuf(p); p = TEXTSTART; }
      } else /* c == 0 */ {
        if (erronly) (void)fprintf(stream, "%6d |", LineNum);
        (void)fputs("(End-of-file)\n", stream);
        p--;
      }
      LineNum++;
    }
  }
  (void)close(fd);

#ifdef MONITOR
  generate_leave ("message");
#endif
}
```

```
#if defined(__cplusplus) || defined(__STDC__)
void
message(int severity, char *Msgtext, int grammar, POSITION *source)
#else
void
message(severity, Msgtext, grammar, source)
int severity; char *Msgtext; int grammar; POSITION *source;
#endif
/* Report an error
 *    On entry-
 *      severity=error severity
 *      Msgtext=message text
 *      grammar=identification of the test that failed
 *      source=source coordinates at which the error was detected
 ***/
{
  int fail = 0;
  struct msg *r, *c;

#ifdef MONITOR
  generate_enter ("message");
  generate_message (key[severity], Msgtext, source->line, source->col);
#endif

  if (severity < NOTE || severity > DEADLY) {
    (void)fprintf(stderr, "Invalid severity code %d for \"%s\"\n",
      severity, Msgtext);
    severity = DEADLY;
    }

  if (source == (POSITION *)0) source = &NoCoord;

  if (ImmediateOutput) {
    (void)fprintf(stderr, "\"%s\", line %d:%d %s: %s",
        source->fn ? source->fn : SRCFILE, source->line,source->col, /*LP*/
    key[severity], Msgtext);
    if (grammar>0 && GrammarLine) (void)fprintf(stderr," AG=%d\n", grammar);
    else (void)putc('\n', stderr);
    (void)fflush(stderr);
  }

  ErrorCount[severity]++;

  if ((r = (struct msg *)malloc(sizeof(struct msg))) == (struct msg *)0) {
    r = &emergency;
    (void)fprintf(stderr, "No storage for error report at");
    fail = 1;
  }
  r->loc = *source;
  r->severity = severity;
  r->Msgtext = Msgtext;
  r->grammar = grammar;

  c = reports.back; while (earlier(&r->loc,&c->loc)) c = c->back;
```

```
  r->forward = c->forward; c->forward = r;
  r->back = c; (r->forward)->back = r;


  if(ErrorLimit &&
     ErrorCount[ERROR] > LineNum/2 +10) { /*LP*/ /* old: LineNum/20 */
    (void)fprintf(stderr, "\"%s\", line %d:%d %s: %s",
        SRCFILE, source->line,source->col, key[DEADLY],
        "Too many ERRORs");
    fail = 1;
  }
  if (severity == DEADLY || fail ) {
    (void)putc('\n', stderr);
#ifdef MONITOR
    generate_leave ("message");
#endif
    exit(1);
  }

#ifdef MONITOR
  generate_leave ("message");
#endif
}
}
```

This macro is attached to an output file.

14.2 Auxiliary scanner for line directives

The auxiliary scanner

68 **auxscan.c**[68] ≡

```
{
/* auxiliary scanner to analyse #line directives */

#include <ctype.h>
#include <string.h>
#include "err.h"
#include "source.h"
#include "gla.h"

extern char* auxEOL (char* start, int length); /* $/Tool/gla/auxScanEOL.c */

char *auxLinedirective (start, length)
  char* start;   /* start of characters recognized by reg expr */
  int length;    /* length of what was recognized already */
{
  char c;
  char *p = start + length; /* first char not yet processed */
  char *startnum,  /* where in p the line number begins */
       *startname, /* where in p the filename begins */
       *endnum,    /* where the 0-terminator must be put after the num */
       *endname;   /* where the 0-terminator must be put (replacing quote) */
  enum { _afterhash, _l, _i, _n, _e, _afterline,
         _num, _afternum, _quote, _name, _finished } state = _afterhash;
  for (;;) {
```

```
    c = *p++;
    if (c == '\0') {  /* refill buffer if necessary */
      refillBuf(p-1);
      p = TEXTSTART;
      StartLine = p-1;
      c = *(p-1);
      if (*p == '\0')
        return (p);
    }
    /* the following is a finite automaton that accepts '  line 123 "asdf"'
       then skips the rest of the line. It stores the beginning and end
       addresses of the number and the filename to be used afterwards.
    */
    if      (state == _afterhash && (c == 9 || c == ' '))
      ;
    else if (state == _afterhash && c == 'l')
      state = _l;
    else if (state == _afterhash && isdigit (c))
      { state = _num; startnum = p-1; }  /* word 'line' may be missing */
    else if (state == _l && c == 'i')
      state = _i;
    else if (state == _i && c == 'n')
      state = _n;
    else if (state == _n && c == 'e')
      state = _e;
    else if (state == _e && (c == 9 || c == ' '))
      state = _afterline;
    else if (state == _afterline && (c == 9 || c == ' '))
      state = _afterline;
    else if (state == _afterline && isdigit (c))
      { state = _num; startnum = p-1; }
    else if (state == _num && isdigit (c))
      ;
    else if (state == _num && (c == 9 || c == ' '))
      { state = _afternum; endnum = p-1; }
    else if (state == _afternum && (c == 9 || c == ' '))
      ;
    else if (state == _afternum && c == '"')
      { state = _name; startname = p; }
    else if (state == _name && c != '"')
      ;
    else if (state == _name && c == '"')
      { state = _finished; endname = p-1; }
    else if (state == _finished && c != '\n')
      ;
    else if (state == _finished && c == '\n')
      break;
    else { /* something went wrong */
      message (ERROR, "this is no #line directive", 0, NoPosition);
      return (auxEOL (start, length));  /* ignore the line */
    }
  }
  *endnum = 0;  /* write 0-terminators */
  *endname = 0;
  curpos.line = LineNum = atoi (startnum);
```

```
  curpos.fn = strdup (startname);
  return (p);
}
#if 0
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;
    else if (state == _ && c == '')
      state = ;

    if (c == '\n') {
      message(ERROR, "incomplete #line directive", 0, &curpos);
      LineNum++;
      StartLine = p-1;
      return(p);
    }
#endif
}
```

This macro is attached to an output file.

PART II: Semantic Analysis

Semantic analysis consists of three parts. The first and smallest is the description of the abstract syntax, the second is consistent renaming for the block-scoped identifiers, and the third is type analysis. The latter is structured in exactly the same fashion as the syntax description above (see the table of contents).

The purpose of type analysis is to compute the properties of all objects[2] in the abstract syntax tree, in particular their types, and to emit error messages for all violations of the semantic rules of the language (except for scoping, which was already checked before).

15 Abstract syntax

To specify the abstract syntax is very simple in Eli: One or several files of type `.sym` specify a set of *nonterminal equivalence classes*. Using these, Eli generates the abstract syntax automatically from the parsing grammar.

For CuPit, most of the equivalence classes are concerned with expressions. The various expression non-terminals for the different precedence levels of operators all belong into the same equivalence class. The same is true for all binary operators.

abstract.sym[69] ≡ 69

```
{
 /* $Id: names.fw,v 1.8 1994/04/13 07:25:50 prechelt Exp prechelt $ */
 Expr    ::=  Expression E1 E2 E3 E4 E5 E6 E7 E8 E9 E10 E11 E12.
 BinOp   ::=  OrOp AndOp CompareOp InOp BitorOp IntervalOp BitandOp
              ShiftOp AddOp MulOp ExponOp.
}
```

This macro is attached to an output file.

That's all, folks!

16 Consistent renaming

The scoping rules of CuPit are similar to those of C: Identifiers can be defined globally (i.e. in the outermost block of the program) or locally (in a nested block). Identifiers must be defined exactly once. Definitions in inner blocks hide definitions in outer blocks, however. This section describes the Eli specifications that solve the problem of assigning a unique symbol value to each definition and use of an identifier so that corresponding definitions and uses get the same symbol; the process is called consistent renaming. Note that the fields of record types (and connection, node, and network types) are not handled the same way, because they obey different scoping rules.

In Eli, library modules are available that define these consistent renaming rules: `Name/Chain.gnrc` describes the visibility of names in blocks, `Name/NoKeyMsg.gnrc` emits an error message for each undefined identifier, and `Name/Unique.gnrc` emits an error message for each multiply defined identifier.

Consistent renaming is an attribution process, so we need a LIDO file to specify it. We call this file `names.lido`:

names.lido[70] ≡ 70

{
 Scoping Basics[71]
 Name Definition[72]
 Name Use[73]

[2] Here we use the word object not for "object in the sense of the CuPit language definition", but for "object handled internally in the compiler"

Name Ranges[74]
}

This macro is attached to an output file.

The scoping basics contain a few basic attribute and symbol definitions. The name definition, use, and ranges macros connect the LIDO symbols that describe identifier definitions, identifier uses, and blocks, respectively, to the corresponding symbols in the grammar.

16.1 Basic scoping rule symbols and attributes

Three LIDO attributes are used in the consistent renaming process: `Env` represents a scope, `Key` holds a pointer to the definition table entry representing an object, and `Sym` is just the identifier number of a name.

The `NameOccurrence` and `TypeNameOccurrence` are just little helpers to pick a name from a small subtree. `IdDef` is an auxiliary symbol that unites all the properties required to describe a name definition: C-like name visibility rules, pick a `LowercaseIdent` as the actual name, and there must at most be one defintion for each name in a block. `IdUse` correspondingly describes name uses: C-like name visibility, pick a `LowercaseIdent` as the actual name, and there must at least be one visible definition for each name. The symbols that are inherited from are defined in the library modules mentioned above.

These auxiliary symbols do not work for the type identifiers, because these are not `LowercaseIdent`s (see below).

71 *Scoping Basics*[71] ≡

```
{
ATTR Env: Environment SYNT;
ATTR Key: DefTableKey SYNT;
ATTR Sym: int SYNT;

SYMBOL NameOccurrence COMPUTE
  SYNT.Sym = CONSTITUENT LowercaseIdent.Sym;
END;
SYMBOL TypeNameOccurrence COMPUTE
  SYNT.Sym = CONSTITUENT UppercaseIdent.Sym;
END;

SYMBOL IdDef INHERITS IdDefChain, NameOccurrence, IdDefUnique END;
SYMBOL IdUse INHERITS IdUseChain, NameOccurrence, NoKeyMsg END;
}
```

This macro is invoked in definition 70.

These auxiliary symbols are now used to connect the scoping rules to the grammar symbols that represent names.

16.2 Name definition, name use, blocks

Name definitions occur in procedure and function definitions, in type definitions (enumeration identifiers!) and in data object definitions. Type names are special, because they are `UppercaseIdent`s.

72 *Name Definition*[72] ≡

```
{
SYMBOL NewTypeId INHERITS IdDefChain, IdDefUnique, TypeNameOccurrence END;
SYMBOL NewEnumId INHERITS IdDef END;
SYMBOL NewDataId INHERITS IdDef END;
SYMBOL NewParamId INHERITS IdDef END;
```

```
SYMBOL NewProcedureId INHERITS IdDef END;
SYMBOL NewFunctionId INHERITS IdDef END;
SYMBOL NewReductionFunctionId INHERITS IdDef END;
SYMBOL NewWtaFunctionId INHERITS IdDef END;
}
```

This macro is invoked in definition 70.

Name uses can occur in expressions, on the left hand side of assignments, in data object definitions (type name use), and in procedure and function calls. Type names are special, because they are `UppercaseIdent`s.

Name Use[73] ≡ 73

```
{
SYMBOL TypeId INHERITS IdUseChain, NoKeyMsg, TypeNameOccurrence END;
SYMBOL Objectname INHERITS IdUse END;
SYMBOL ProcedureId INHERITS IdUse END;
SYMBOL FunctionId INHERITS IdUse END;
SYMBOL ReductionFunctionId INHERITS IdUse END;
SYMBOL WtaFunctionId INHERITS IdUse END;
}
```

This macro is invoked in definition 70.

To distinguish scopes, we have to define what the range of visibility for a name is. The root symbol is a special range, because it implicitly includes the definitions of the predefined symbols.

Name Ranges[74] ≡ 74

```
{
SYMBOL CupitProgram INHERITS RootChain, RangeUnique
COMPUTE
  SYNT.Env = StandardEnv (NewEnv());
END;
SYMBOL Statements INHERITS RangeChain, RangeUnique END;
SYMBOL SubroutineDescription INHERITS RangeChain, RangeUnique END;
}
```

This macro is invoked in definition 70.

16.3 Predefined identifiers

The objects that are predefined in CuPit are the basic types and the boolean value constants true and false. In addition, we need a predefined key for the type `Void` that is used only internally in the compiler:

Predefined Objects[75] ≡ 75

```
{
PredefObj (Bool,   UppercaseIdent)
PredefObj (Int,    UppercaseIdent)
PredefObj (Int1,   UppercaseIdent)
PredefObj (Int2,   UppercaseIdent)
PredefObj (Real,   UppercaseIdent)
PredefObj (String, UppercaseIdent)

PredefObj (Interval,   UppercaseIdent)
PredefObj (Interval1,  UppercaseIdent)
PredefObj (Interval2,  UppercaseIdent)
PredefObj (Realerval,  UppercaseIdent)

PredefObj (true,  LowercaseIdent)
```

```
PredefObj (false, LowercaseIdent)

PredefKey (Void);
}
```

This macro is invoked in definitions 76, 77, and 79.

The way these predefined objects are handled in the compiler is the following: For each of the objects, a variable is declared that holds a definition table key pointing to the objects property definitions (which will be added in later modules). Then we have a procedure called `StandardEnv` that creates an environment consisting of all the predefined objects; this procedure is called to compute the `Env` attribute of the `CupitProgram` symbol.

There is one additional standard objects for which a key is computed. This key, however, is not bound to any identifier, which means that it can not be accessed from the user program; instead, it is used for internal handling only: `Void` is used for the handling of procedures; procedures are assigned the virtual return type `Void` so that they can be treated more like functions.

Now for the implementation of these predefined objects: First we need to define the variables for the object keys. For this purpose, we create a header file `scope.h` that contains the appropriate `extern` declarations and an implementation file `scope.c` that contains the corresponding definitions. In both cases, we use the above declarations by declaring appropriate `PredefObj` and `PredefKey` preprocessor macros.

76 **scope.h**[76] ≡

```
{
#ifndef scope_H
#define scope_H

#include "deftbl.h"
#include "envmod.h"
#include "pdl_gen.h"

#define PredefObj(name,type)  extern DefTableKey name##Key;
#define PredefKey(name)       extern DefTableKey name##Key;
Predefined Objects[75]
#undef PredefObj
#undef PredefKey

extern Environment StandardEnv (Environment e);

#endif
}
```

This macro is attached to an output file.

The `scope.c` file does also contain the implementation of the `StandardEnv` function.

77 **scope.c**[77] ≡

```
{
#include "scope.h"
#include "termcode.h"

#define PredefObj(name,type)  DefTableKey name##Key;
#define PredefKey(name)       DefTableKey name##Key;
Predefined Objects[75]
#undef PredefObj
#undef PredefKey

StandardEnv function[79]
```

}

This macro is attached to an output file.

The contents of the header file are not only needed to compile `scope.c`, but also to compile the LIGA-generated attribution module, so we also stuff it into a corresponding `scope.head` file:

scope.head[78] ≡ 78

```
{
#include "scope.h"
}
```

This macro is attached to an output file.

What is left to do is the actual `StandardEnv` procedure. Once again, clever preprocessor macros do most of the work:

StandardEnv function[79] ≡ 79

```
{
Environment StandardEnv(Environment e)
{
  /* Add the predefined objects to Environment e
     This PredefObj macro works only for ANSI-C preprocessors
  */
  int Code, idn;
#define PredefObj(name,type)\
Code = type; \
mkidn(#name, strlen (#name), &Code, &idn); \
name##Key = DefineIdn (e, idn);\
SetSym (name##Key, idn, idn);
#define PredefKey(name)   name##Key = NewKey();
  Predefined Objects[75]
#undef PredefObj
#undef PredefKey
  return e;
}
}
```

This macro is invoked in definition 77.

17 General type analysis definitions

17.1 Kinds of objects

The properties that have to be determined for an object differ from one category of objects to the other. We will call this category of an object its *Kind*. We define the following kinds:

Kinds Of Objects[80] ≡ 80

```
{
typedef enum {
  UndefinedK,
  SimpleTypeK,    SymbolicTypeK,  RecordTypeK,    ConTypeK,       NodeTypeK,
  NetTypeK,       ArrayTypeK,     NodeArrayTypeK, NodeGroupTypeK,
  ProcedureK,     FunctionK,
  ReductionFunctionK,             WtaFunctionK,   MergeProcedureK,
  VariableK,      ParVariableK,   ParVariableSelK,ConstantK,
  ErrorK
} tKind;
```

```
}
```

This macro is invoked in definition 195.

`UndefinedK` is the kind of an object that is not declared and about which nothing is known.

The `xTypeK` kinds are explained in respect to figure 1: A `SimpleTypeK` object is one of the simple types; a `RecordTypeK` object is either a record type or an interval type; a `ConTypeK` object is a connection type; a `NodeTypeK` object is a node type, node array type, or node group type; a `NetTypeK` object is a network type; an `ArrayTypeK` object is an array type whose base type is neither a connection or network type (which are forbidden) nor a node type (which is covered by `NodeArrayTypeK`). `NodeGroupTypeK` is (surprise! surprise!) the kind of a node group type.

A `ProcedureK` or `FunctionK` object is a global, extern, or object procedure or function, respectively. A `ReductionFunctionK` or `WtaFunctionK` object is a reduction function or winner-takes-all function, respectively.

A `VariableK` object is either an object declared as either `VAR, CONST`, or `IO` or an expression whose value cannot be determined at compile-time; a `ConstantK` object is either a constant (denoter), constant expression, or a `CONST` object, whose value can be determined at compile time; a `ConstantK` can have simple type or interval type. A `ParVariableK` is a `VariableK` object that represents a parallel variable: not only a single object of its type, but a multitude of objects on which we can operate in parallel. Parallel variables are node arrays (having `NodeArrayTypeK` type), node groups (having `NodeGroupTypeK` type), slices of node arrays or groups (having `NodeTypeK` type), and the connection interfaces in a node (having `ConTypeK` type). Network variables are *not* parallel variables, although they might induce parallel operations due to existing replicates. Parallel variables are relevant for selection, subscription, I/O assignment, and object subroutine call.

The `ErrorK` kind is used to flag internal errors in the compiler only.

17.2 Properties and attributes

To store the properties of objects in the definition table, these properties have to be declared in a tiny language called property definition language in Eli. This language allows to include header files to declare types, to give a list of property names and a type name to declare properties and to request additional operations (beyond the standard `GetX` and `SetX`) to be defined for individual properties. Here is the property definition file that is used for the type analysis in the CuPit compiler:

81 *Type Analysis Properties*[81] ≡

```
{
 Sym:          Int;
 Kind:         tKind;
 Type:         DefTableKey;
 OilType:      tOilType;
 Access:       tAccess;
 Mode:         tInterfaceMode;
 Dataloc:      tInterfaceMode;
 IsUsed:       int;
 MergeDefs:    int;
 Val:          CupitConst;
 Params:       KeyArray;
 CentralAgent: Bool;
 MayBeRead,
 MayBeWritten: DefTblKeySet;
 NeedConReduction,
 NeedNodeReduction,
 NeedNetReduction,
 NeedConWta,
 NeedNodeWta,
```

```
NeedNetWta:   Bool;
}
```

This macro is invoked in definition 192.

Most properties correspond to attributes with the same name. The semantics of the attributes and properties are discussed below.

In the now following attribute declarations, the keywords `SYNT` and `INH` indicate that the respective attributes are synthesized (i.e., propagated upwards in the syntax tree) or inherited (i.e., propagated downwards in the syntax tree), respectively. Eli could usually determine this attribute class itself; the information is given here explicitly for increased clarity.

Type Analysis Attributes[82] ≡ 82

```
{
ATTR Sym:              int SYNT;
ATTR nr:               int SYNT;
ATTR str:              String SYNT;
ATTR Kind,    Kind2:   tKind SYNT;
ATTR InhKind:          tKind INH;
ATTR Key:              DefTableKey SYNT;
ATTR Type,    Type2:   DefTableKey SYNT;
ATTR ParVarType:       DefTableKey SYNT;
ATTR InhType, InhKey:  DefTableKey INH;
ATTR MeType,  YouType: DefTableKey INH;
ATTR ScopeKey:         DefTableKey INH;
ATTR Access:           tAccess SYNT;
ATTR InhAccess:        tAccess INH;
ATTR Mode:             tInterfaceMode SYNT;
ATTR ParVarMode:       tInterfaceMode SYNT;
ATTR InhMode:          tInterfaceMode INH;
ATTR Dataloc:          tInterfaceMode SYNT;
ATTR IsUsed:           int SYNT;
ATTR Context:          tContext INH;
ATTR LoopNest:         int INH;
ATTR Val:              CupitConst SYNT;
ATTR Params:           KeyArray SYNT;
ATTR InhParams:        KeyArray INH;
ATTR CentralAgent:     Bool SYNT;
ATTR MayBeUsed:        DefTblKeySet SYNT;
ATTR MayBeRead:        DefTblKeySet SYNT;
ATTR MayBeWritten:     DefTblKeySet SYNT;
ATTR NetVar:           DefTableKey SYNT;
ATTR Operator:         tOilOp SYNT;
ATTR done1, done2:     VOID SYNT;
}
```

This macro is invoked in definition 193.

The `Sym` attribute is the string handle returned by the scanner for all non-literal terminal symbols (`TERM`s). `nr` and `str` are use-it-as-you-like attributes that are used as local variables for various purposes in some rules.

The meaning of `Kind` was already discussed above. The `InhKind` attribute is used to propagate the type kind of a declaration list into the list body (containing the names).

The `Type` and `Key` attributes are used to propagate type information and object identity information, respectively, upwards in the tree. `InhType` and `InhKey` are used to propagate a `Type` or `Key` that came upwards from one subtree downwards into another (for example to give type definition bodies access to the type name they describe, so properties of the type name can be set from rules for the type definition body). This `InhX` naming convention is used for many attributes that are used in a way similar to this; these attributes will not always be explained individually. `Type2` is an auxiliary attribute used locally in some rules.

The `Type` property and attribute is used to store the type of data objects, formal parameters, and expressions and the return type of global functions, object functions, and reduction functions. For procedures a virtual return type represented by `VoidKey` is used. For array and group types, `Type` is used to store the base type. `MeType` represents the type of the implicit parameter `ME` for object procedures, object functions, reduction functions, winner-takes-all functions, and merge procedures. Analogously, `YouType` represents the type of the implicit parameter `YOU` for reduction functions, winner-takes-all functions, and merge procedures. The property `OilType` is used to store the corresponding Oil type for each named type; it is not set for data objects. See section 22.1. `ParVarType` propagates the type of a parallel variable to a parallel variable selection, because the type is needed for the code generation of `CONNECT` and `REDUCTION` statements.

`Key` represents the definition table entry created for each named object. The definition table module is created automatically by Eli from the `pdl` files. Due to interface constraints of the generic name analysis module `field.gnrc`, `Key` must also be used inherited in some places. `ScopeKey` is introduced locally by the `field.gnrc` module; we declare it globally here for convenience and clarity.

The `Access` property specifies for a named data object or a formal parameter whether it has `CONST`, `VAR`, or `IO` access rights (values `ConstAcc`, `VarAcc`, `IoAcc`, respectively). For objects that are parameters, a corresponding set of values also announces this fact (values `ConstPAcc`, `VarPAcc`, `IoPAcc`, respectively). The `Access` attribute reflects the same for expressions. `ME` and `YOU` are always treated as parameters because they are of course implemented this way.

83 *Access Rights of Objects*[83] ≡

```
{
typedef enum {
  NoAcc, ConstAcc, VarAcc, IoAcc, ConstPAcc, VarPAcc, IoPAcc
} tAccess;
}
```

This macro is invoked in definition 195.

The following macros are supplied for testing `tAccess` values and to convert normal access values into parameter access values:

84 *Parameter Access Handler*[84] ≡

```
{
#define IsConstAcc(x) ((x) == ConstAcc || (x) == ConstPAcc)
#define IsVarAcc(x)   ((x) == VarAcc   || (x) == VarPAcc)
#define IsIoAcc(x)    ((x) == IoAcc    || (x) == IoPAcc)
#define ParameterAcc(x) ((x) == ConstAcc ? ConstPAcc : \
                         (x) == VarAcc   ? VarPAcc :   \
                         (x) == IoAcc    ? IoPAcc :    NoAcc)
}
```

This macro is invoked in definition 196.

The `Mode` property specifies for a connection interface element in a node type whether it is an `IN` or an `OUT` interface (values `InMode` or `OutMode`, respectively).

85 *Interface Modes of Connection Elements*[85] ≡

```
{
typedef enum {
  NoMode, InMode, OutMode
} tInterfaceMode;
}
```

This macro is invoked in definition 195.

`ParVarMode` propagates the mode of a parallel variable to a parallel variable selection, because the value is needed for the code generation of, for instance, `REDUCTION` statements.

The `Dataloc` property specifies for a connection type, which data location was chosen for it: `InMode` means the data is located at input interfaces of nodes (i.e. at the output end of the connection) and the output interfaces of this type hold only remote connection objects; `OutMode` means the opposite situation.

The `IsUsed` property specifies for each procedure and function whether and how it is used in the program: `used0` means the subroutine is not used at all (i.e. no code has to be generated), `used` means the subroutine is used sequentially (for plain subroutines, meaning the sequential version has to be generated) or from the same level of parallelism (for object subroutines, meaning the unvirtualized version has to be generated), `usedA` means the subroutine is called in parallel (for plain subroutines, meaning the parallel version has to be generated) or is used to introduce additional parallelism (for object procedures, meaning the virtualized version must be generated), `usedAR` is valid for connection procedures only and means a `usedA`-type call for remote connections (meaning the remote virtualized version has to be generated; as opposed to a `usedA`-type call for local connections). The property is bitcoded. For calls from global procedures that are not part of the central agent, we set both `used` and `usedA`.
IsUsed Values[86] ≡ 86

```
{
 #define used0   0
 #define used    1
 #define usedA   2
 #define usedAR  4
 }
```

This macro is invoked in definition 196.

The `Context` attribute is used to discriminate the kinds of environments for data declarations and statements:

Kinds Of Contexts[87] ≡ 87

```
{
 typedef enum {
   NoContext, GlobalContext, GlobalSubroutineContext, TypeDefContext,
   ObjSubroutineContext, ReductionContext, WtaContext, MergeContext
 } tContext;
 }
```

This macro is invoked in definition 195.

The `LoopNest` attribute describes the current nesting level of loops in order to be able to correctly reject a `BREAK` statement outside of any loop. The root context has nesting level 0.

The `MergeDefs` property is used to store the number of `MERGE` procedures seen for a structure type; zero or one are legal, more than one is an error.

The `Val` property is used to store the values of constant objects and constant expressions of integer, real, or interval types. The `Val` property is also used to store the number of elements of an array type. The `Val` attribute propagates the values during the evaluation of constant expressions. When `Val` is invalid, it holds the value `ErrorConst`.

The `Params` attribute and property are used to compute and store the parameter list of a procedure or function. `InhParams` is used to supply the formal parameter list for the analysis of an actual parameter list. The `Params` property is also used for types to store the arguments needed by a type constructor.

The `CentralAgent` attribute and property are used to compute and whether a certain global subroutine is free (false) or belongs to the central agent (true). The attribute is computed at object names (true for network variables, false otherwise) and global subroutine calls (true for central agent subroutines, false otherwise) and collected at the subroutine definition level where it is stored in the property of the subroutine; due to the defined-before-applied restriction on the order of subroutine definitions, this behavior constitutes a correct transitive-hull computation.

The `MayBeUsed`, `MayBeRead`, and `MayBeWritten` attributes are used to compute for each object subroutine the elements of `ME` that are statically read or written in it (i.e. *may* be dynamically read or written in a call to it). `MayBeUsed` is used at the object and expression level where we can not decide whether the access is a read or write access. The other two attributes are used at the statement level, for subscription expressions, and for actual parameter expressions where we can decide whether the accesses are read or write accesses. The corresponding properties are set for each object subroutine.

The `NetVar` attribute is computed at the `InitDataId` symbol and is the key for network variables and `NoKey` otherwise. This symbol is used with a CONSTITUENTS clause at the root symbol in order to collect the names of the variables that have to be initialized by a global initialization procedure call.

The `Operator` attribute is used to propagate the Oil representation of an operator from an operator leaf to the expression subtree immediately above; see section 22.1.

`NeedConReduction`, `NeedNodeReduction`, and `NeedNetReduction` are boolean properties that are set to true for the type `x` when an actual reduction statement for values of type `x` is encountered during type analysis. Which of the three properties is used depends on whether the reduction statement is found in a connection, node or network subroutine. The properties announce that an `a_REDUCTION` procedure has to be generated for the respective object category. The properties are reset to false by the code generation phase as soon as the `a_REDUCTION` procedure was generated. The property remains undefined for any type for which no respective reduction statement is issued somewhere in the program.

`NeedConWta`, `NeedNodeWta`, and `NeedNetWta` are the analog properties for winner-takes-all functions (`a_WTA`). They are used in exactly the same fashion.

`done1` and `done2` are used locally in some rules merely to introduce dependencies for certain actions.

17.3 Properties of predefined objects

For the predefined objects, the properties must be set by the compiler before the actual compilation begins. This is done in a way similar to that used to define the predefined objects in section 16.3.

For the predefined types, we only have to set the `Kind` property and compute the corresponding Oil types, for the predefined constants (`true, false`), the kind, type and value must be set.

88 *Set Properties of Predefined Objects*[88] ≡

```
{
 extern void SetPredefObjProperties ()
 {
    DefTableKey  HelpKey;
    KeyArray     HelpArray;
    SetKind (BoolKey, SimpleTypeK, ErrorK);
    SetKind (IntKey,  SimpleTypeK, ErrorK);
    SetKind (Int1Key, SimpleTypeK, ErrorK);
    SetKind (Int2Key, SimpleTypeK, ErrorK);
    SetKind (RealKey, SimpleTypeK, ErrorK);
    SetKind (StringKey, SimpleTypeK, ErrorK);

    SetKind (IntervalKey,  RecordTypeK, ErrorK);
    SetKind (Interval1Key, RecordTypeK, ErrorK);
    SetKind (Interval2Key, RecordTypeK, ErrorK);
    SetKind (RealervalKey, RecordTypeK, ErrorK);

    SetKind (trueKey, ConstantK, ErrorK);
    SetVal  (trueKey, SetIval (true), ErrorConst);
    SetType (trueKey, BoolKey, NoKey);
    SetKind (falseKey, ConstantK, ErrorK);
    SetVal  (falseKey, SetIval (false), ErrorConst);
```

```
    SetType (falseKey, BoolKey, NoKey);

    Set Oil Types For Standard Types[148]

    Define Conversion[89](`Int',` Real')
    Define Conversion[89](`Int2',` Int')
    Define Conversion[89](`Int1',` Int')
    Define Conversion[89](`Real',` Int')
}
}
```
This macro is invoked in definition 194.

For the standard type conversions we define the `Params` property with a single constant parameter given.

Define Conversion[89](◇2) ≡ 89

```
{HelpArray = NewKeyArray (1);
 HelpKey = NewKey();
 SetAccess (HelpKey, ConstPAcc, NoAcc);
 SetType   (HelpKey, ◇2Key, NoKey);
 StoreKeyInArray (HelpArray, 0, HelpKey);
 SetParams (◇1Key, HelpArray, NoKeyArray);
 }
```
This macro is invoked in definitions 88, 88, 88, and 88.

17.4 General traversal order

Most of the CuPit type analysis can be done in a single text-order traversal of the syntax tree. We thus define a `CHAIN known` that represents the invariant "all visible objects have all their visible type analysis properties defined".

The declaration of a `CHAIN x` makes Eli introduce two attributes `x_pre` and `x_post` of `VOID` type (i.e., the attributes are either set or not, but do not have any particular value). Accessing a chain always accesses either of these attributes: In a rule context `A ::= B C` assignments `A.x = ...` and uses `B.x` access `x_post` while assignments `B.x = ...` or `C.x = ...` and uses `A.x` access `x_pre`. These attribute accesses can be used to force computations in arbitrary parts of the subtree below the `CHAINSTART` of the chain to occur in a single left-to-right depth-first text-order traversal of the subtree.

Traversal Order Invariant[90] ≡ 90

```
{
 CHAIN known: VOID;
 SYMBOL CupitProgram COMPUTE
   CHAINSTART HEAD.known =
     ORDER (SetPredefObjProperties (),
            Messag (NOTE, "setpredefobjproperties()")) DEPENDS_ON THIS.Env;
   Messag (NOTE, "everything is 'known' now  ") DEPENDS_ON TAIL.known;
   IF (GT (NrOfErrors, 0),
     Message (DEADLY, "No code was generated")) DEPENDS_ON TAIL.known;
 END;
 }
```
This macro is invoked in definition 193.

18 Type definitions

Type Definition Analysis[91] ≡ 91

{
Type Definition Key[92]
Symbolic Type Analysis[94]
Structure Type Scoping Rules[95]
Record Type Analysis[96]
Node Type Analysis[98]
Connection Type Analysis[100]
Array Type Analysis[101]
Group Type Analysis[102]
Network Type Analysis[103]
}

This macro is invoked in definition 193.

When a type definition is processed, two things have to be done:

1. The structure of the new type must be determined and a (compiler-internal) object that describes this structure be created.
2. The structure of the new type must be bound to the name of the new type.

The latter is prepared here by making the `Key` of the new type name available to the type definition body subtree `TypeDefBody` as `InhKey`. The same technique is used at many other places in this compiler; it will not be described there again.

92 *Type Definition Key*[92] ≡

```
{
RULE rTypeDef :
  TypeDef ::=  'TYPE' NewTypeId 'IS' TypeDefBody 'END' OptTYPE
COMPUTE
  TypeDefBody.InhKey = NewTypeId.Key;
  TypeDefBody.Context = TypeDefContext;
END;
}
```

This macro is defined in definitions 92 and 93.
This macro is invoked in definition 91.

For all identifiers we set the `Sym` property in order to be able to produce error messages that explicitly contain the identifier although at the point where the message is produced we only have its key. This is particularly useful for type names in order to make type conflicts clearer in the error messages.

93 *Type Definition Key*[93] ≡

```
{
RULE rNewTypeId :
  NewTypeId ::=  UppercaseIdent
COMPUTE
  NewTypeId.known = SetSym (NewTypeId.Key, NewTypeId.Sym, NoSym)
  DEPENDS_ON NewTypeId.known;
END;
}
```

This macro is defined in definitions 92 and 93.
This macro is invoked in definition 91.

18.1 Symbolic type definitions

To introduce a symbolic type, we have to compute and bind a value (the `EnumIdNo`) for each enumeration identifier. Since assignment and comparison is defined for values of a symbolic type, we also instantiate an Oil schema (using `NewSymbolicOilType`) for the type.

94 *Symbolic Type Analysis*[94] ≡

```
{
CHAIN EnumIdNo : int;
RULE rSymbolicTypeDef :
  SymbolicTypeDef ::=  'SYMBOLIC' NewEnumIdList OptSEMICOLON
COMPUTE
  .Type = INCLUDING TypeDefBody.InhKey;
  NewEnumIdList.known =
    ORDER (SetKind (.Type, SymbolicTypeK, SymbolicTypeK),
           SetOilType (.Type, NewSymbolicOilType(.Type), OilErrorType()))
  DEPENDS_ON SymbolicTypeDef.known;
  CHAINSTART NewEnumIdList.EnumIdNo = 0;
END;

RULE rNewEnumId :
  NewEnumId ::=  LowercaseIdent
COMPUTE
  NewEnumId.known =
     ORDER (SetVal  (NewEnumId.Key, SetIval (NewEnumId.EnumIdNo), ErrorConst),
            SetKind (NewEnumId.Key, ConstantK, ErrorK),
            SetSym  (NewEnumId.Key, NewEnumId.Sym, NoSym))
  DEPENDS_ON NewEnumId.known;
  NewEnumId.EnumIdNo = ADD (NewEnumId.EnumIdNo, 1); /* postincrement */
END;
}
```

This macro is invoked in definition 91.

18.2 Scoping in structured types

For the elements of structured types (records, connections, nodes, networks) different scoping rules apply than for the other names in a CuPit program: If an element name is defined within a structure of type `T`, then the scope of that definition is the name following the dot in all phrases of the form `Object '.' LowercaseIdent` for which `Object` yields an object of type `T`.

Scope rules of this kind are common in programming languages, and they cannot be verified during scope analysis because they depend on type analysis. Eli provides a generic library module `$/Tool/lib/Name/Field.gnrc` to implement consistent renaming according to this rule. `Field.gnrc` exports four symbols to describe the concepts involved: A `FieldScope` is a phrase that contains element definitions, a `FieldDef` is an element name definition, and a `FieldUse` is an element name use. `RootField` is a phrase containing all of the `FieldScope` phrases in the source program. Name definitions and uses are assumed to be represented by tree nodes having a `Sym` attribute that specifies the corresponding name (this attribute is automatically set by the scanner in Eli). The module will compute the value of a `Key` attribute of type `DefTableKey` at each tree node representing a name definition or use. `Key` attribute values of associated definitions and uses will be identical. If a use is not associated with any definition, its `Key` attribute will be the distinguished `DefTableKey` value `NoKey` (which is always used in Eli to indicate a non-valid or unavailable key).

Each `FieldScope` must be provided with a `Key` attribute of type `DefTableKey` by some mechanism outside of the module. The same `DefTableKey` value must be provided as the `ScopeKey` attribute of each `FieldUse` or `FieldDef`, again via a mechanism outside of the module. It is this `DefTableKey` value that links the field with the record in which it is defined.

Since the element names must be unique within a structure and every element must be defined, the symbols also inherit from the error reporting modules discussed in section 16.

The rules for `FieldUse` will not be defined here; they are discussed in section 23.

Structure Type Scoping Rules[95] ≡ 95

```
{
 SYMBOL CupitProgram INHERITS RootField END;
 SYMBOL StructureTypeDef: Key: DefTableKey INH;
 SYMBOL StructureTypeDef INHERITS FieldScope, RangeUnique COMPUTE
   INH.Key = INCLUDING TypeDefBody.InhKey;  /* to satisfy Field.gnrc */
 END;

 SYMBOL NewElemId INHERITS FieldDef, IdDefUnique, NameOccurrence COMPUTE
   INH.ScopeKey = INCLUDING TypeDefBody.InhKey; /* to satisfy Field.gnrc */
   SYNT.known =
     ORDER (
       SetType (THIS.Key, INCLUDING InitElemIdList.InhType, NoKey),
       SetKind (THIS.Key, VariableK, VariableK),
       SetSym  (THIS.Key, THIS.Sym, NoSym),
       Messag3 (NOTE, "%s %s into type %s",
                  SymString (GetSym (INCLUDING InitElemIdList.InhType, NoSym)),
                  SymString (THIS.Sym),
                  SymString (GetSym (THIS.ScopeKey, NoSym))))
     DEPENDS_ON THIS.known;
 END;
 }
```

This macro is invoked in definition 91.

18.3 Record type definitions

We set the `Kind` property of a record type to `RecordTypeK`. For each data element in a record type definition we have to compute its type property. Therefore, for any `RecordDataElemDef` phrase we propagate the type given to be accessed by the `NewElemId` phrases.

96 *Record Type Analysis*[96] ≡

```
{
 SYMBOL RecordTypeDef INHERITS StructureTypeDef END;

 RULE rRecordTypeDef :
   RecordTypeDef ::=  'RECORD' RecordElemDefList
 COMPUTE
   .Type = RecordTypeDef.Key;
   RecordElemDefList.known = ORDER (
     SetKind (.Type, RecordTypeK, RecordTypeK),
     Messag1 (NOTE, "RecordScope %s", SymString (GetSym (.Type, NoSym))))
   DEPENDS_ON RecordTypeDef.known;
 END;

 RULE rRecordDataElemDef :
   RecordDataElemDef ::=  TypeId InitElemIdList
 COMPUTE
   InitElemIdList.InhType = TypeId.Key;
   InitElemIdList.InhKind = TypeId.Kind;
   InitElemIdList.known =
     IF (EQ (TypeId.Kind, NetTypeK),
       Message (ERROR, "networks cannot be elements of records"),
     IF (OR (OR (EQ (TypeId.Kind, NodeTypeK), EQ (TypeId.Kind, NodeArrayTypeK)),
         EQ (TypeId.Kind, NodeGroupTypeK)),
       Message (ERROR, "nodes cannot be elements of records"),
     IF (EQ (TypeId.Kind, ConTypeK),
```

```
        Message (ERROR, "connections cannot be elements of records"))))
      DEPENDS_ON RecordDataElemDef.known;
    END;
    }
```

This macro is defined in definitions 96 and 97.
This macro is invoked in definition 91.

The now following analysis of the "initialized element identifier list" (`InitElemIdList`) is also used for the data elements of the network, node, and connection types. We just record the type and kind (`VariableK`) of each element and check that the initializer (if any) has an assignment-compatible type and a constant value.

Record Type Analysis[97] ≡ 97

```
  {
    RULE rInitElemIdList :
      InitElemIdList ::=  InitElemIdList ',' InitElemId
    COMPUTE
      TRANSFER InhType, InhKind;
    END;

    RULE rInitElemIdList1 :
      InitElemIdList ::=  InitElemId
    COMPUTE
      TRANSFER InhType, InhKind;
    END;

    RULE rInitElemId1 :
      InitElemId ::=  NewElemId ':=' Expr
    COMPUTE
      InitElemId.known =
        ORDER (
          /* for SetKind, SetType see SYMBOL NewElemId */
          IF (AND (AND (NE (InitElemId.InhType, NoKey), /* Elem type defined */
                  NE (Expr.Type, NoKey)),        /* and initializer type defined */
                  NOT (OilIsValidOp (OilIdOp2 (AssignOp,   /* but assignment not */
                   DefTbl2Oil (InitElemId.InhType), DefTbl2Oil (Expr.Type))))),
            Message2 (ERROR, "Initializer has wrong type: %s (expected: %s)",
                      SymString (GetSym (Expr.Type, NoSym)),
                      SymString (GetSym (InitElemId.InhType, NoSym)))))
        DEPENDS_ON TAIL.known;
    END;
    }
```

This macro is defined in definitions 96 and 97.
This macro is invoked in definition 91.

For definitions of subroutine elements (procedures, merge procedures, functions), see section 20.

18.4 Node type definitions

The scoping rules for node elements are the same as for record elements; see section 18.2 for a description. See section 18.3 for a description of the analysis of data element definition lists. The same element types are allowed; the messages for illegal element types are different, though. In addition to the element categories for record types we have to process interface definitions.

Node Type Analysis[98] ≡ 98

```
  {
```

```
SYMBOL NodeTypeDef INHERITS StructureTypeDef END;

RULE rNodeTypeDef :
  NodeTypeDef ::=  'NODE' NodeElemDefList
COMPUTE
  .Type = NodeTypeDef.Key;
  NodeElemDefList.known = ORDER (
    SetKind (.Type, NodeTypeK, NodeTypeK),
    Messag1 (NOTE, "NodeScope %s", SymString (GetSym (.Type, NoSym))))
  DEPENDS_ON NodeTypeDef.known;
END;

RULE rNodeDataElemDef :
  NodeDataElemDef ::=  TypeId InitElemIdList
COMPUTE
  InitElemIdList.InhType = TypeId.Key;
  InitElemIdList.InhKind = TypeId.Kind;
  InitElemIdList.known =
    IF (EQ (TypeId.Kind, NetTypeK),
      Message (ERROR, "Networks cannot be elements of nodes"),
    IF (OR (OR (EQ (TypeId.Kind, NodeTypeK), EQ (TypeId.Kind, NodeArrayTypeK)),
        EQ (TypeId.Kind, NodeGroupTypeK)),
      Message (ERROR, "Nodes cannot be elements of nodes"),
    IF (EQ (TypeId.Kind, ConTypeK),
      Message (ERROR, "Interface mode (IN or OUT) missing"))))
  DEPENDS_ON NodeDataElemDef.known;
END;
}
```

This macro is defined in definitions 98 and 99.
This macro is invoked in definition 91.

For interface definitions (**NodeInterfaceElemDef**) we have to check that the given type is a connection type.

99 *Node Type Analysis*[99] ≡

```
{
RULE rNodeInterfaceElemDef :
  NodeInterfaceElemDef ::=  InterfaceMode TypeId InterfaceIdList
COMPUTE
  InterfaceIdList.InhMode = InterfaceMode.Mode;
  InterfaceIdList.InhType = TypeId.Key;
  InterfaceIdList.InhKind = TypeId.Kind;
  InterfaceMode.known =
    IF (NE (GetKind (TypeId.Key, ConTypeK), ConTypeK),
      Message (ERROR, "Only connection types can have IN or OUT"))
  DEPENDS_ON NodeInterfaceElemDef.known;
END;

RULE rInterfaceModeIn :
  InterfaceMode ::=  'IN'
COMPUTE
  InterfaceMode.Mode = InMode;
END;

RULE rInterfaceModeOut :
  InterfaceMode ::=  'OUT'
COMPUTE
```

```
  InterfaceMode.Mode = OutMode;
END;

RULE rInterfaceIdList1 :
  InterfaceIdList ::=  NewInterfaceId
COMPUTE
  TRANSFER InhType, InhKind, InhMode;
END;

RULE rInterfaceIdList :
  InterfaceIdList ::=  InterfaceIdList ',' NewInterfaceId
COMPUTE
  TRANSFER InhType, InhKind, InhMode;
END;

SYMBOL NewInterfaceId INHERITS FieldDef, IdDefUnique, NameOccurrence COMPUTE
  INH.ScopeKey = INCLUDING TypeDefBody.InhKey; /* to satisfy Field.gnrc */
  SYNT.known =
    ORDER (SetKind (THIS.Key, VariableK, VariableK),
           SetType (THIS.Key, THIS.InhType, NoKey),
           SetSym  (THIS.Key, THIS.Sym, NoSym),
           SetMode (THIS.Key, THIS.InhMode, NoMode)) DEPENDS_ON THIS.known;
END;
}
```

This macro is defined in definitions 98 and 99.
This macro is invoked in definition 91.

18.5 Connection type definitions

The scoping rules for connection elements are the same as for record elements; see section 18.2 for a description. See section 18.3 for a description of the analysis of data element definition lists. The same element types are allowed; the messages for illegal element types are different, though.

Connection Type Analysis[100] ≡ 100

```
{
SYMBOL ConnectionTypeDef INHERITS StructureTypeDef END;

RULE rConnectionTypeDef :
  ConnectionTypeDef ::=  'CONNECTION' ConElemDefList
COMPUTE
  .Type = ConnectionTypeDef.Key;
  ConElemDefList.known = ORDER (
    SetKind (.Type, ConTypeK, ConTypeK),
    SetDataloc (.Type, IF (conAtOut, OutMode, InMode), NoMode),
    Messag1 (NOTE, "ConScope %s", SymString (GetSym (.Type, NoSym))))
  DEPENDS_ON ConnectionTypeDef.known;
END;

RULE rConDataElemDef :
  ConDataElemDef ::=  TypeId InitElemIdList
COMPUTE
  InitElemIdList.InhType = TypeId.Key;
  InitElemIdList.InhKind = TypeId.Kind;
  InitElemIdList.known =
    IF (EQ (TypeId.Kind, NetTypeK),
```

```
      Message (ERROR, "Networks cannot be elements of connections"),
    IF (OR (OR (EQ (TypeId.Kind, NodeTypeK), EQ (TypeId.Kind, NodeArrayTypeK)),
        EQ (TypeId.Kind, NodeGroupTypeK)),
      Message (ERROR, "Nodes cannot be elements of connections"),
    IF (EQ (TypeId.Kind, ConTypeK),
      Message (ERROR, "Connections cannot be elements of connections"))))
  DEPENDS_ON ConDataElemDef.known;
END;
}
```

This macro is defined in definitions 100.
This macro is invoked in definition 91.

18.6 Array type definitions

An array type can have any other type as its base type, except a connection or network type. We thus have to emit an error message for these cases. Negative or non-integral or non-constant array sizes also evoke an error message.

101 *Array Type Analysis*[101] ≡

```
{
ATTR BaseTypeKind, ArrayTypeKind : tKind;

RULE rArrayTypeDef :
  ArrayTypeDef ::=  'ARRAY' '[' ArraySize ']' 'OF' TypeId
COMPUTE
  .BaseTypeKind = TypeId.Kind;
  .ArrayTypeKind =
    IF (EQ (.BaseTypeKind, ConTypeK),
        ORDER (Message (ERROR, "Arrays of connections are impossible"),
               UndefinedK),
    /* else: */
    IF (EQ (.BaseTypeKind, NetTypeK),
        ORDER (Message (ERROR, "Arrays of networks are not allowed"),
               UndefinedK),
    /* else: */
    IF (OR (OR (EQ (.BaseTypeKind, ArrayTypeK),
        EQ (.BaseTypeKind, NodeArrayTypeK)),
        EQ (.BaseTypeKind, NodeGroupTypeK)),
        ORDER (Message (ERROR, "Arrays of arrays are not implemented"),
               UndefinedK),
    /* else: */
    IF (EQ (.BaseTypeKind, NodeTypeK),
        NodeArrayTypeK,
    /* else: */
    ArrayTypeK)))) DEPENDS_ON ArrayTypeDef.known;
  ArrayTypeDef.known =
    ORDER (.ArrayTypeKind,
           SetKind (INCLUDING TypeDefBody.InhKey, .ArrayTypeKind,
                                                  .ArrayTypeKind),
           SetType (INCLUDING TypeDefBody.InhKey, TypeId.Key, NoKey),
           SetVal  (INCLUDING TypeDefBody.InhKey, ArraySize.Val, ErrorConst))
    DEPENDS_ON TAIL.known;
END;

RULE rArraySize :
```

```
  ArraySize ::=  Expr
COMPUTE
  ArraySize.Val =
    IF (AND (NE (Expr.Kind, UndefinedK), NE (Expr.Kind, ConstantK)),
        ORDER (Message (ERROR, "Array size must be constant expression"),
               SetIval (1)),
    /* else */
    IF (AND (AND (AND (NE (Expr.Type, IntKey), NE (Expr.Type, Int2Key)),
        NE (Expr.Type, Int1Key)), NE (Expr.Type, NoKey)),
        ORDER (Message (ERROR, "Array size must be integer expression"),
               SetIval (1)),
    /* else */
    IF (AND (EQ (Expr.Kind, ConstantK), LE (GetIval (Expr.Val), 0)),
        ORDER (Message1 (ERROR, "Array size must be positive (is: %d)",
                         GetIval (Expr.Val)),
               SetIval (1)),
    /* else */
    Expr.Val))) DEPENDS_ON TAIL.known;
  ArraySize.known = ArraySize.Val;
END;
}
```

This macro is invoked in definition 91.

18.7 Group type definitions

A group type must have a node type as its base type; we thus have to emit an error message if this is not the case.

Group Type Analysis[102] ≡ 102

```
{
ATTR BaseTypeKind, GroupTypeKind : tKind;

RULE rGroupTypeDef :
  GroupTypeDef ::=  'GROUP' 'OF' TypeId
COMPUTE
  .BaseTypeKind = TypeId.Kind;
  .GroupTypeKind =
    IF (AND (NE (.BaseTypeKind, UndefinedK), NE (.BaseTypeKind, NodeTypeK)),
        ORDER (Message (ERROR, "Groups can be defined for Nodes only"),
               UndefinedK),
        NodeGroupTypeK);
  GroupTypeDef.known =
    ORDER (SetKind (INCLUDING TypeDefBody.InhKey, .GroupTypeKind,
                                              .GroupTypeKind),
           SetType (INCLUDING TypeDefBody.InhKey, TypeId.Key, NoKey))
    DEPENDS_ON TypeId.known;
END;
}
```

This macro is invoked in definition 91.

18.8 Network type definitions

The scoping rules for network elements are the same as for record elements; see section 18.2 for a description. Node groups and node arrays are allowed as elements.

103 *Network Type Analysis*[103] ≡

```
{
SYMBOL NetworkTypeDef INHERITS StructureTypeDef END;

RULE rNetworkTypeDef :
  NetworkTypeDef ::=  'NETWORK' NetElemDefList
COMPUTE
  .Type = NetworkTypeDef.Key;
  NetElemDefList.known = ORDER (
    SetKind (.Type, NetTypeK, NetTypeK),
    Messag1 (NOTE, "NetScope %s", SymString (GetSym (.Type, NoSym))))
  DEPENDS_ON NetworkTypeDef.known;
END;

RULE rNetDataElemDef :
  NetDataElemDef ::=  TypeId InitElemIdList
COMPUTE
  InitElemIdList.InhType = TypeId.Key;
  InitElemIdList.InhKind = TypeId.Kind;
  InitElemIdList.known =
    IF (EQ (TypeId.Kind, NetTypeK),
      Message (ERROR, "Networks cannot be elements of Networks"),
    IF (EQ (TypeId.Kind, NodeTypeK),
      Message (ERROR, "Single nodes cannot be elements of networks"),
    IF (EQ (TypeId.Kind, ConTypeK),
      Message (ERROR, "Connections cannot be explicit elements of networks"))))
  DEPENDS_ON NetDataElemDef.known;
END;
}
```

This macro is defined in definitions 103.
This macro is invoked in definition 91.

19 Data object definitions

To handle data object definitions, we have to propagate the type and access information into the `InitDataIdList`. For the latter, we then have to check types compatibility and correct presence or absence of initializers.

104 *Data Object Definition Analysis*[104] ≡

```
{
RULE rDataObjectDef:
  DataObjectDef ::=  TypeId AccessType InitDataIdList
COMPUTE
  InitDataIdList.InhType = TypeId.Key;
  InitDataIdList.InhKind = TypeId.Kind;
  InitDataIdList.InhAccess = AccessType.Access;
END;

RULE rTypeId :
  TypeId ::= UppercaseIdent
COMPUTE
  TypeId.Kind = GetKind (TypeId.Key, UndefinedK) DEPENDS_ON TypeId.known;
  TypeId.known = TypeId.Kind;
END;
```

```
RULE rConstAccess :
  AccessType ::=  'CONST'
COMPUTE
  AccessType.Access = ConstAcc;
END;

RULE rVarAccess :
  AccessType ::=  'VAR'
COMPUTE
  AccessType.Access = VarAcc;
END;

RULE rIoAccess :
  AccessType ::=  'IO'
COMPUTE
  AccessType.Access = IoAcc;
END;
}
```

This macro is defined in definitions 104 and 105.
This macro is invoked in definition 193.

With the declarations above, the type and access of the declaration are available in the `InitDataIdList`. So now we check the following: (1) Initializers may be present at `VAR` objects, (2) must be present at `CONST` objects, and (3) must not be present at `IO` objects. (4) If an initializer is present, its type must be assignment compatible with the object type; for global variables and constants, this intializer must be a compile-time constant expression. We also compute the `Kind`: For `CONST` objects with a constant expression as initializer it is `ConstantK`, otherwise it is `VariableK`.

Data Object Definition Analysis[105] ≡ 105

```
{
RULE rInitDataIdList1 :
  InitDataIdList ::=  InitDataId
COMPUTE
  TRANSFER InhType, InhKind, InhAccess;
END;

RULE rInitDataIdList :
  InitDataIdList ::=  InitDataIdList ',' InitDataId
COMPUTE
  TRANSFER InhType, InhKind, InhAccess;
END;

RULE rInitDataId0 :
  InitDataId ::=  NewDataId
COMPUTE
  InitDataId.NetVar =
    IF (EQ (InitDataId.InhKind, NetTypeK),
      NewDataId.Key, /* else */ NoKey);
  NewDataId.known =
    IF (IsConstAcc (InitDataId.InhAccess),
      Message (ERROR, "CONST objects must be initialized"))
    DEPENDS_ON InitDataId.known;
END;

RULE rInitDataId1 :
  InitDataId ::=  NewDataId ':=' Expr
```

```
COMPUTE
  .Type = InitDataId.InhType;
  InitDataId.NetVar =
    IF (EQ (InitDataId.InhKind, NetTypeK),
      NewDataId.Key, /* else */ NoKey);
  InitDataId.known = ORDER (
    IF (IsIoAcc (InitDataId.InhAccess),
      Message (ERROR, "IO objects cannot be initialized")),
    IF (AND (IsConstAcc (InitDataId.InhAccess),
             EQ (Expr.Kind, ConstantK)),
      ORDER (SetKind (NewDataId.Key, ErrorK, ConstantK),  /* is a change! */
             SetVal (NewDataId.Key, Expr.Val, ErrorConst),
             Messag2 (NOTE, "CONST %s = %f",
                      SymString (GetSym (NewDataId.Key, NoSym)),
                      GetRval (GetVal (NewDataId.Key, ErrorConst))))),
    IF (AND (EQ (Context Kind[111], GlobalContext),
             EQ (Expr.Kind, VariableK)),
      Message (ERROR, "Initializers of global CONSTs/VARs must be constant")),
    IF (/* Incompatible type of initializer: */
        AND (AND (NE (.Type, NoKey), NE (Expr.Type, NoKey)),
        NOT (OilIsValidOp (OilIdOp2 (AssignOp, DefTbl2Oil (.Type),
                                     DefTbl2Oil(Expr.Type))))),
      Message2 (ERROR, "Initializer has incompatible type: %s (expected: %s)",
                SymString (GetSym (Expr.Type, NoSym)),
                SymString (GetSym (.Type, NoSym)))))
  DEPENDS_ON Expr.known;
END;

RULE rNewDataId :
  NewDataId ::=  LowercaseIdent
COMPUTE
  NewDataId.known =
    ORDER (SetType (NewDataId.Key, INCLUDING InitDataId.InhType, NoKey),
           SetKind (NewDataId.Key, VariableK, VariableK),
           SetSym  (NewDataId.Key, NewDataId.Sym, NoSym),
           SetAccess (NewDataId.Key, INCLUDING InitDataId.InhAccess, VarAcc))
    DEPENDS_ON NewDataId.known;
END;
}
```

This macro is defined in definitions 104 and 105.
This macro is invoked in definition 193.

20 Subroutine definitions

The analysis of subroutine definitions must determine the following features:

1. Result type of subroutine (`Void` for procedures); stored in `InhType` attribute and in `Type` property of subroutine name.
2. Object type of subroutine (`Void` for non-object subroutines); stored in `MeType` attribute.
3. Type and access mode of all parameters; stored in `Params` property.
4. Initial local environment of procedure body, based on parameter list.

The analysis consists of the following parts:

106 *Subroutine Definition Analysis*[106] ≡

```
{
 Subroutine Type Remote Access[107]
 Procedure Definition Analysis[112]
 Function Definition Analysis[113]
 Object Procedure Definition Analysis[114]
 Object Function Definition Analysis[116]
 Parameterlist Definition Analysis[117]
 Reduction Function Definition Analysis[121]
 Wta Function Definition Analysis[122]
 Merge Procedure Definition Analysis[123]
 }
```

This macro is invoked in definition 193.

For easy access (via INCLUDING) to the `InhType` attribute from procedures and to `MeType` and `YouType` from non-object procedures, we assign the value `Void` for these attributes at the root symbol and define appropriate macros for the access. A similar technique is used to make the `Context` accessible globally.

Subroutine Type Remote Access[107] ≡ 107

```
{
 SYMBOL CupitProgram COMPUTE
   INH.InhType = VoidKey DEPENDS_ON THIS.Env;
   INH.Context = GlobalContext;
 END;
 }
```

This macro is invoked in definition 106.

The following three macros can be used to access the result type or the type of the `ME` or `YOU` object from within a subroutine body:

Subroutine Return Type[108] ≡ 108

```
{INCLUDING (
   CupitProgram.InhType,
   SubroutineDescription.InhType,
   ReductionFunctionBody.InhType,
   WtaFunctionBody.InhType)
 }
```

This macro is invoked in definitions 134, 134, 134, 134, and 134.

Subroutine ME Type[109] ≡ 109

```
{INCLUDING (
   CupitProgram.InhType,
   TypeDefBody.InhKey,
   ReductionFunctionBody.MeType,
   WtaFunctionBody.MeType,
   MergeProcedureBody.MeType)
 }
```

This macro is invoked in definitions 112, 113, 115, 130, 131, 131, 131, 132, 138, 139, 182, 186, 187, and 188.

Subroutine YOU Type[110] ≡ 110

```
{INCLUDING (
   CupitProgram.InhType,
   ReductionFunctionBody.YouType,
   WtaFunctionBody.YouType,
   MergeProcedureBody.YouType)
 }
```

This macro is invoked in definition 186.

To determine the context of a statement or declaration we use the following similar construction:

111 *Context Kind*[111] ≡
```
{INCLUDING (
   CupitProgram.Context,
   TypeDefBody.Context,
   SubroutineDescription.Context,
   ReductionFunctionBody.Context,
   WtaFunctionBody.Context,
   MergeProcedureBody.Context)
 }
```
This macro is invoked in definitions 105, 128, 128, 131, 132, 138, 139, 143, 177, 182, 184, 184, 186, 186, and 187.

20.1 Normal procedures and functions

The simplest case for subroutine definition analysis are ordinary global procedures: Set the `Type` property of the procedure name to `Void`, inherit the procedure name key into the procedure body and parameter list analysis using the `InhKey` attribute, and store `ProcedureK` in the `Kind` property of the procedure name. The `SubroutineDescription` phrase (including the parameter list) is handled in section 20.3 below.

112 *Procedure Definition Analysis*[112] ≡
```
{
 RULE rProcedureDef :
   ProcedureDef ::=  'PROCEDURE' NewProcedureId SubroutineDescription
                     OptPROCEDURE
 COMPUTE
   SubroutineDescription.known =
     SetType (NewProcedureId.Key, VoidKey, NoKey)
     DEPENDS_ON NewProcedureId.known;
   SubroutineDescription.InhKey  = NewProcedureId.Key;
   SubroutineDescription.InhType = VoidKey;
   SubroutineDescription.Context =
     IF (EQ (Subroutine ME Type[109], VoidKey),
       GlobalSubroutineContext, /* else */ ObjSubroutineContext);
   ProcedureDef.CentralAgent =
     CONSTITUENTS (FunctionCall.CentralAgent, ProcedureCall.CentralAgent,
                   Objectname.CentralAgent)
                  WITH (Bool, OR, IDENTICAL, BoolNull)
     DEPENDS_ON SubroutineDescription.known;
   ProcedureDef.known =
     IF (EQ (SubroutineDescription.Context, GlobalSubroutineContext),
       SetCentralAgent (NewProcedureId.Key, ProcedureDef.CentralAgent, false))
     DEPENDS_ON SubroutineDescription.known;
 END;

 RULE rNewProcedureId :
   NewProcedureId ::=  LowercaseIdent
 COMPUTE
   NewProcedureId.known = ORDER (
       SetKind (NewProcedureId.Key, ProcedureK, ProcedureK),
       SetSym  (NewProcedureId.Key, NewProcedureId.Sym, NoSym))
     DEPENDS_ON NewProcedureId.known;
 END;
 }
```
This macro is invoked in definition 106.

Global functions are handled analogously to global procedures, except that we have to register their return type in the `Type` property of the function name and in the `InhType` attribute of the function description phrase:

Function Definition Analysis[113] ≡ 113

```
{
RULE rFunctionDef :
  FunctionDef ::=  TypeId 'FUNCTION' NewFunctionId SubroutineDescription
                   OptFUNCTION
COMPUTE
  SubroutineDescription.known =
    SetType (NewFunctionId.Key, TypeId.Key, NoKey)
    DEPENDS_ON NewFunctionId.known;
  SubroutineDescription.InhKey  = NewFunctionId.Key;
  SubroutineDescription.InhType = TypeId.Key;
  SubroutineDescription.Context =
    IF (EQ (Subroutine ME Type[109], VoidKey),
      GlobalSubroutineContext, /* else */ ObjSubroutineContext);
  FunctionDef.CentralAgent =
    CONSTITUENTS (ProcedureCall.CentralAgent, FunctionCall.CentralAgent,
                  Objectname.CentralAgent)
                 WITH (Bool, OR, IDENTICAL, BoolNull)
    DEPENDS_ON SubroutineDescription.known;
  FunctionDef.known =
    IF (EQ (SubroutineDescription.Context, GlobalSubroutineContext),
      SetCentralAgent (NewFunctionId.Key, FunctionDef.CentralAgent, false))
    DEPENDS_ON SubroutineDescription.known;
END;

RULE rNewFunctionId :
  NewFunctionId ::=  LowercaseIdent
COMPUTE
  NewFunctionId.known = ORDER (
      SetKind (NewFunctionId.Key, FunctionK, FunctionK),
      SetSym (NewFunctionId.Key, NewFunctionId.Sym, NoSym))
    DEPENDS_ON NewFunctionId.known;
END;
}
```

This macro is invoked in definition 106.

20.2 Object procedures and functions

For object subroutines, we define a `SYMBOL NewObjRoutineId` that makes it easier to satisfy the interface requirements of the `Field.gnrc` module for the introduction of field names (in this case: subroutine names).

Object Procedure Definition Analysis[114] ≡ 114

```
{
SYMBOL NewObjRoutineId INHERITS FieldDef, IdDefUnique, NameOccurrence COMPUTE
  INH.ScopeKey = INCLUDING TypeDefBody.InhKey; /* to satisfy Field.gnrc */
END;
}
```

This macro is defined in definitions 114 and 115.
This macro is invoked in definition 106.

Using this symbol, we now compute the type attributes and properties for the procedure as a whole (in analogy to the way we treat global procedures in the previous section):

115 *Object Procedure Definition Analysis*[115] ≡

```
{
 SYMBOL NewObjProcedureId INHERITS NewObjRoutineId END;

 RULE rObjProcedureDef :
   ObjProcedureDef ::=  'PROCEDURE' NewObjProcedureId SubroutineDescription
                        OptPROCEDURE
 COMPUTE
   .Kind = GetKind (Subroutine ME Type[109], UndefinedK)
           DEPENDS_ON ObjProcedureDef.known;
   SubroutineDescription.known =
     SetType (NewObjProcedureId.Key, VoidKey, NoKey)
     DEPENDS_ON NewObjProcedureId.known;
   SubroutineDescription.InhType = VoidKey;
   SubroutineDescription.InhKey  = NewObjProcedureId.Key;
   SubroutineDescription.Context = ObjSubroutineContext;
   ObjProcedureDef.MayBeRead =
     SubroutineDescription CONSTITUENTS (Assignment.MayBeRead,
       InputAssignment.MayBeRead, OutputAssignment.MayBeRead,
       ProcedureCall.MayBeRead, ObjectProcedureCall.MayBeRead,
       ReductionStmt.MayBeRead, WtaStmt.MayBeRead,
       ReturnStmt.MayBeRead, IfStmt.MayBeRead, LoopStmt.MayBeRead,
       DataAllocationStmt.MayBeRead)
     WITH (DefTblKeySet, DSunite, IDENTICAL, DSempty);
   ObjProcedureDef.MayBeWritten =
     SubroutineDescription CONSTITUENTS (Assignment.MayBeWritten,
       InputAssignment.MayBeWritten,
       ProcedureCall.MayBeWritten, ObjectProcedureCall.MayBeWritten,
       ReductionStmt.MayBeWritten, WtaStmt.MayBeWritten,
       LoopStmt.MayBeWritten, DataAllocationStmt.MayBeWritten)
     WITH (DefTblKeySet, DSunite, IDENTICAL, DSempty);
   /* !!! ORDER (
     Message (NOTE, ""),
     DSprint (stderr, ObjProcedureDef.MayBeRead),
     DSprint (stderr, ObjProcedureDef.MayBeWritten),
     fprintf (stderr, " is read/written in %s\n",
              SymString (NewObjProcedureId.Sym))); */
   ObjProcedureDef.known = ORDER (
     SetMayBeRead (NewObjProcedureId.Key, ObjProcedureDef.MayBeRead,
                   NoDefTblKeySet),
     SetMayBeWritten (NewObjProcedureId.Key, ObjProcedureDef.MayBeWritten,
                      NoDefTblKeySet))
     DEPENDS_ON TAIL.known;
 END;

 RULE rNewObjProcedureId :
   NewObjProcedureId ::=  LowercaseIdent
 COMPUTE
   NewObjProcedureId.known = ORDER (
       SetKind (NewObjProcedureId.Key, ProcedureK, ProcedureK),
       SetSym  (NewObjProcedureId.Key, NewObjProcedureId.Sym, NoSym))
     DEPENDS_ON NewObjProcedureId.known;
 END;
```

```
}
```
This macro is defined in definitions 114 and 115.
This macro is invoked in definition 106.

The analysis of object functions works similarly, but here we have to set the function result type attribute `InhKey` as well:

Object Function Definition Analysis[116] ≡ 116

```
{
SYMBOL NewObjFunctionId INHERITS NewObjRoutineId END;

RULE rObjFunctionDef :
  ObjFunctionDef ::=  TypeId 'FUNCTION' NewObjFunctionId SubroutineDescription
                      OptFUNCTION
COMPUTE
  SubroutineDescription.known =
    SetType (NewObjFunctionId.Key, TypeId.Key, NoKey)
    DEPENDS_ON NewObjFunctionId.known;
  SubroutineDescription.InhKey  = NewObjFunctionId.Key;
  SubroutineDescription.InhType = TypeId.Key;
  SubroutineDescription.Context = ObjSubroutineContext;
END;

RULE rNewObjFunctionId :
  NewObjFunctionId ::=  LowercaseIdent
COMPUTE
  NewObjFunctionId.known = ORDER (
      SetKind (NewObjFunctionId.Key, FunctionK, FunctionK),
      SetSym  (NewObjFunctionId.Key, NewObjFunctionId.Sym, NoSym))
    DEPENDS_ON NewObjFunctionId.known;
END;
}
```
This macro is defined in definitions 116.
This macro is invoked in definition 106.

20.3 Parameter lists

Parameter lists are represented as objects of type `KeyArray` and stored in the `Params` attribute and property. To collect the parameters and give each one a number (used to select the position in the `KeyArray`), we define a `CHAIN Paramcounter` of integer type. The fact that the number of a particular parameter identifier has been computed, is indicated by a void attribute `GotParam`.

Parameterlist Definition Analysis[117] ≡ 117

```
{
CHAIN Paramcounter : int;
SYMBOL NewParamId:  GotParam: VOID;

RULE rSubroutineDescription :
  SubroutineDescription ::=  ParamList 'IS' SubroutineBody 'END'
COMPUTE
  CHAINSTART ParamList.Paramcounter = 0;
  SubroutineBody.known =
    SetParams (SubroutineDescription.InhKey, ParamList.Params, ParamList.Params)
    DEPENDS_ON (ParamList CONSTITUENTS NewParamId.GotParam,
                ParamList.known);
END;
```

```
RULE rSubroutineDescription0 :
  SubroutineDescription ::=  ParamList 'IS' 'EXTERNAL'
COMPUTE
  CHAINSTART ParamList.Paramcounter = 0;
  SubroutineDescription.known =
    SetParams (SubroutineDescription.InhKey, ParamList.Params, ParamList.Params)
    DEPENDS_ON (ParamList CONSTITUENTS NewParamId.GotParam,
                ParamList.known);
END;
}
```

This macro is defined in definitions 117, 118, 119, and 120.
This macro is invoked in definition 106.

When the attribution of a parameter list starts, we first have to count the parameters, then allocate a `KeyArray` of the appropriate size, and then put all the individual parameters in this key array. The process is simplified when the parameter list is empty.

118 *Parameterlist Definition Analysis*[118] ≡

```
{
RULE rParamList0 :
  ParamList ::=  '(' ')'
COMPUTE
  ParamList.Params = NewKeyArray (0);
END;

RULE rParamList :
  ParamList ::=  '(' Parameters ')'
COMPUTE
  ParamList.Params = NewKeyArray (Parameters.Paramcounter);
END;
}
```

This macro is defined in definitions 117, 118, 119, and 120.
This macro is invoked in definition 106.

For each parameter, we have to compute the type and the access; these are propagated down to each `NewParamId` using inherited attributes:

119 *Parameterlist Definition Analysis*[119] ≡

```
{
RULE rParamsDef :
  ParamsDef ::=  TypeId AccessType ParamIdList
COMPUTE
  ParamIdList.InhType = TypeId.Key;
  ParamIdList.InhKind = TypeId.Kind;
  ParamIdList.InhAccess = ParameterAcc (AccessType.Access);
END;

RULE rParamIdList1 :
  ParamIdList ::=  NewParamId
COMPUTE
  TRANSFER InhType, InhKind, InhAccess;
END;

RULE rParamIdList :
  ParamIdList ::=  ParamIdList ',' NewParamId
COMPUTE
```

```
  TRANSFER InhType, InhKind, InhAccess;
END;
}
```

This macro is defined in definitions 117, 118, 119, and 120.
This macro is invoked in definition 106.

All parameters have `VariableK` kind, since we do not perform interprocedural constant propagation analysis. When the parameter is stored in the key array, the `GotParam` attribute is set; this must happen only after all properties for the parameter were computed.

Parameterlist Definition Analysis[120] ≡ 120

```
{
RULE rNewParamId :
  NewParamId ::=  LowercaseIdent
COMPUTE
  NewParamId.known =
    ORDER (SetType (NewParamId.Key, NewParamId.InhType, NoKey),
           SetKind (NewParamId.Key, VariableK, VariableK),
           SetSym  (NewParamId.Key, NewParamId.Sym, NoSym),
           SetAccess (NewParamId.Key, NewParamId.InhAccess, VarPAcc))
    DEPENDS_ON NewParamId.known;
  NewParamId.GotParam =
    ORDER (StoreKeyInArray (INCLUDING ParamList.Params,
                            NewParamId.Paramcounter, NewParamId.Key),
           Messag3 (NOTE, "Param %d: %s %s", NewParamId.Paramcounter,
                    SymString (GetSym (NewParamId.InhType, NoSym)),
                    SymString (GetSym (NewParamId.Key, NoSym))))
    DEPENDS_ON NewParamId.known;
  NewParamId.Paramcounter = ADD (NewParamId.Paramcounter, 1);
END;
}
```

This macro is defined in definitions 117, 118, 119, and 120.
This macro is invoked in definition 106.

20.4 Reduction functions

Reduction functions and winner-takes-all functions have implicit parameters `ME` and `YOU` whose types can be seen from the type name given in the function definition. For reduction functions this is also the result type; for winner-takes-all functions the result type is always `Bool`. The context of the body of a reduction function is `ReductionContext`, for a winner-takes-all function it is `WtaContext`.

Reduction Function Definition Analysis[121] ≡ 121

```
{
RULE rReductionFunctionDef :
  ReductionFunctionDef ::=  TypeId 'REDUCTION' NewReductionFunctionId 'IS'
                            ReductionFunctionBody 'END' OptREDUCTION
COMPUTE
  ReductionFunctionBody.known =
    SetType (NewReductionFunctionId.Key, TypeId.Key, NoKey)
    DEPENDS_ON NewReductionFunctionId.known;
  ReductionFunctionBody.InhType = TypeId.Key;
  ReductionFunctionBody.MeType  = TypeId.Key;
  ReductionFunctionBody.YouType = TypeId.Key;
  ReductionFunctionBody.Context = ReductionContext;
END;
```

```
RULE rNewReductionFunctionId :
  NewReductionFunctionId ::=  LowercaseIdent
COMPUTE
  NewReductionFunctionId.known = ORDER (
      SetKind (NewReductionFunctionId.Key, ReductionFunctionK,
                                            ReductionFunctionK),
      SetSym (NewReductionFunctionId.Key, NewReductionFunctionId.Sym, NoSym))
    DEPENDS_ON NewReductionFunctionId.known;
END;
}
```

This macro is invoked in definition 106.

20.5 Winner-takes-all functions

(see description under reduction functions above)

122 *Wta Function Definition Analysis*[122] ≡

```
{
RULE rWtaFunctionDef :
  WtaFunctionDef ::=  TypeId 'WTA' NewWtaFunctionId 'IS'
                      WtaFunctionBody 'END' OptWTA
COMPUTE
  WtaFunctionBody.known =
    SetType (NewWtaFunctionId.Key, BoolKey, NoKey)
    DEPENDS_ON NewWtaFunctionId.known;
  NewWtaFunctionId.InhType = TypeId.Key;
  WtaFunctionBody.InhType = BoolKey;
  WtaFunctionBody.MeType  = TypeId.Key;
  WtaFunctionBody.YouType = TypeId.Key;
  WtaFunctionBody.Context = WtaContext;
END;

RULE rNewWtaFunctionId :
  NewWtaFunctionId ::=  LowercaseIdent
COMPUTE
  NewWtaFunctionId.known = ORDER (
      SetKind (NewWtaFunctionId.Key, WtaFunctionK, WtaFunctionK),
      SetType (NewWtaFunctionId.Key, NewWtaFunctionId.InhType, NoKey),
      SetSym  (NewWtaFunctionId.Key, NewWtaFunctionId.Sym, NoSym))
    DEPENDS_ON NewWtaFunctionId.known;
END;
}
```

This macro is invoked in definition 106.

20.6 Merge procedures

Since Merge procedures have no name, no assignment of name properties or attributes is necessary. However, the object type has to be propagated into the merge procedure body and the context type `MergeContext` must be indicated. We also count the merge procedure definition in the type's `MergeDefs` property:

123 *Merge Procedure Definition Analysis*[123] ≡

```
{
RULE rMergeProcDef :
```

```
  MergeProcDef ::=  'MERGE' 'IS' MergeProcedureBody 'END' OptMERGE
COMPUTE
  .Type = INCLUDING TypeDefBody.InhKey;
  .Kind = GetKind (.Type, ErrorK)   /* Kind of Type ! */
    DEPENDS_ON MergeProcDef.known;
  MergeProcedureBody.MeType  = .Type;
  MergeProcedureBody.YouType = .Type;
  MergeProcedureBody.Context = MergeContext;
  MergeProcedureBody.known =
    ORDER (
      SetMergeDefs (.Type, 1, ADD (GetMergeDefs (.Type, 0), 1)),
      IF (GT (GetMergeDefs (.Type, 0), 1),
        Message (ERROR, "Only one MERGE procedure allowed per type")),
      IF (EQ (.Kind, RecordTypeK),
        Message (ERROR, "MERGE procedures are impossible in RECORD types")))
    DEPENDS_ON MergeProcDef.known;
END;
}
```

This macro is invoked in definition 106.

21 Statements

The semantic analysis for statements consists of the following parts, which will be discussed in order in a separate section each:

Statement Analysis[124] ≡ 124

{
Assignment Analysis[125]
I/O Assignment Analysis[127]
Procedure Call Analysis[128]
Reduction Statement Analysis[131]
Winner-takes-all Analysis[132]
Control Flow Analysis[133]
Data allocation Analysis[138]
Merge Statement Analysis[143]
}

This macro is invoked in definition 193.

21.1 Assignment

For an assignment we have to check that (1) the object assigned to is a plain variable or variable parameter, (2) the types of the variable and the expression are compatible, and (3) we like the shape of the programmer's nose.

Assignment Analysis[125] ≡ 125

```
{
RULE rAssignment :
  Assignment ::=  Object AssignOperator Expr
COMPUTE
  .Operator = AssignOperator.Operator;
  GETSUBCOORD (2);
  Assignment.MayBeRead =
    DSunite (Object.MayBeRead, DSunite (Expr.MayBeRead, Expr.MayBeUsed));
```

```
  Assignment.MayBeWritten = Object.MayBeUsed;
  Assignment.known =
    ORDER (
      IF (IsIoAcc (Object.Access),
        Message (ERROR, "Use  <nodes> --> <IO>  for assignment to IO objects"),
      /* else */
      IF (IsIoAcc (Expr.Access),
      Message (ERROR, "Use  <nodes> <-- <IO>  for assignment from IO objects"),
      /* else */
      IF (AND (NE (Object.Kind, UndefinedK), NOT (IsVarAcc (Object.Access))),
        Message (ERROR, "Object to be assigned to is no variable"),
      /* else */
      IF (EQ (Object.Kind, ParVariableK),
        Message (ERROR, "Can't assign to parallel variable"),
      /* else */
      IF (AND (AND (NE (Object.Type, NoKey), NE (Expr.Type, NoKey)),
          NOT (OilIsValidOp (OilIdOp2 (.Operator, DefTbl2Oil (Object.Type),
                                       DefTbl2Oil(Expr.Type))))),
        Message2 (ERROR, "Object and value incompatible in assignment (%s/%s)",
                  SymString (GetSym (Object.Type, NoSym)),
                  SymString (GetSym (Expr.Type, NoSym))))))))
    DEPENDS_ON Expr.known;
END;
}
```

This macro is defined in definitions 125 and 126.
This macro is invoked in definition 124.

The type checks for assignments are done using Oil, see page 108 for the corresponding Oil declarations.

126 *Assignment Analysis*[126] ≡

{
Operator Mapping[172](‘`AssignOp`’,‘ `AssignOperator`’,‘ `':='`’)
Operator Mapping[172](‘`PlusAssignOp`’,‘ `AssignOperator`’,‘ `'+='`’)
Operator Mapping[172](‘`MinusAssignOp`’,‘ `AssignOperator`’,‘ `'-='`’)
Operator Mapping[172](‘`MulAssignOp`’,‘ `AssignOperator`’,‘ `'*='`’)
Operator Mapping[172](‘`DivAssignOp`’,‘ `AssignOperator`’,‘ `'/='`’)
Operator Mapping[172](‘`ModAssignOp`’,‘ `AssignOperator`’,‘ `'%='`’)
}

This macro is defined in definitions 125 and 126.
This macro is invoked in definition 124.

21.2 I/O assignment

For input and output assignments we have to check that the left hand side object represents a data field of a slice of a node group or node array (a `ParVariableSelK`) and the right hand side object is an `IO` object of the same type as the data field. We set the type attribute for the input and output assignment statements so that the code generation can easily determine for which types a corresponding procedure has to be generated.

127 *I/O Assignment Analysis*[127] ≡

```
{
 RULE rInputAssignment :
   InputAssignment ::=  Object '<--' Object
 COMPUTE
   InputAssignment.known = ORDER (
     IF (NE (Object[1].Kind, ParVariableSelK),
```

```
      Message (ERROR, "Must be parallel variable selection"),
    IF (NOT (IsIoAcc (Object[2].Access)),
      Message (ERROR, "Right-hand-side must be IO object"))),
    IF (AND (AND (NE (Object[1].Type, NoKey), NE (Object[2].Type, NoKey)),
        NE (Object[1].Type, Object[2].Type)),
      Message2 (ERROR, "Type conflict %s/%s: types must be identical",
                SymString (GetSym (Object[1].Type, NoSym)),
                SymString (GetSym (Object[2].Type, NoSym)))))
    DEPENDS_ON TAIL.known;
  InputAssignment.MayBeRead    = Object[1].MayBeRead;
  InputAssignment.MayBeWritten = Object[1].MayBeUsed;
  InputAssignment.Type = Object[2].Type;
END;

RULE rOutputAssignment :
  OutputAssignment ::=  Object '-->' Object
COMPUTE
  OutputAssignment.known = ORDER (
    IF (NE (Object[1].Kind, ParVariableSelK),
      Message (ERROR, "Must be parallel variable selection"),
    IF (NOT (IsIoAcc (Object[2].Access)),
      Message (ERROR, "Right-hand-side must be IO object"))),
    IF (AND (AND (NE (Object[1].Type, NoKey), NE (Object[2].Type, NoKey)),
        NE (Object[1].Type, Object[2].Type)),
      Message2 (ERROR, "Type conflict %s/%s: types must be identical",
                SymString (GetSym (Object[1].Type, NoSym)),
                SymString (GetSym (Object[2].Type, NoSym)))))
    DEPENDS_ON TAIL.known;
  OutputAssignment.MayBeRead = DSunite (Object[1].MayBeRead,
                                        Object[1].MayBeUsed);
  OutputAssignment.Type = Object[2].Type;
END;
}
```

This macro is invoked in definition 124.

21.3 Procedure call

Most of the work for procedure calls is the checking of the actual parameters against the formal parameter list. Most of this is done in the rules for `ExprList`; in the rule for the procedure call itself we only check that the parameter list has the correct length. We use the same `CHAIN Paramcounter` as for the processing of formal parameter lists (see section 20.3).

The only other thing to check is that the meant-to-be procedure identifier really is a procedure identifier.

Procedure Call Analysis[128] ≡ 128

```
{
RULE rProcedureCall :
  ProcedureCall ::=  ProcedureId '(' ArgumentList ')'
COMPUTE
  CHAINSTART ArgumentList.Paramcounter = 0;
  ArgumentList.InhParams = GetParams (ProcedureId.Key, NoKeyArray)
    DEPENDS_ON ProcedureCall.known;
  .nr = KeyArraySize (ArgumentList.InhParams);
  ProcedureCall.known =
    IF (AND (EQ (ProcedureId.Kind, ProcedureK),
```

```
        NE (.nr, ArgumentList.Paramcounter)),
        Message2 (ERROR, "Procedure call has %d parameters, expected: %d",
                  ArgumentList.Paramcounter, .nr))
    DEPENDS_ON (TAIL.known, .done1, .done2);
  ProcedureCall.CentralAgent =
    GetCentralAgent (ProcedureId.Key, false) DEPENDS_ON ProcedureCall.known;
  ProcedureCall.MayBeRead    = ArgumentList.MayBeRead;
  ProcedureCall.MayBeWritten = ArgumentList.MayBeWritten;
  .done1 =
    IF (AND (ProcedureCall.CentralAgent,
        NE (Context Kind[111], GlobalSubroutineContext)),
      Message1 (ERROR, "%s belongs to central agent, cannot be called here",
                SymString (ProcedureId.Sym)));
  .IsUsed =
    IF (EQ (Context Kind[111], GlobalSubroutineContext),
      BITOR(used,usedA), /*else*/ usedA);
  .done2 =
    SetIsUsed (ProcedureId.Key, .IsUsed,
               BITOR (GetIsUsed (ProcedureId.Key,used0), .IsUsed));
END;

RULE rProcedureId :
  ProcedureId ::=  LowercaseIdent
COMPUTE
  ProcedureId.Kind = GetKind (ProcedureId.Key, UndefinedK)
    DEPENDS_ON ProcedureId.known;
  ProcedureId.known =
    IF (EQ (ProcedureId.Kind, FunctionK),
      Message (ERROR, "Result of function call not used"),
    /* else */
    IF (NE (GetKind (ProcedureId.Key, ProcedureK), ProcedureK),
      Message1 (ERROR, "%s is called as a global procedure but it is none",
                SymString (ProcedureId.Sym))));
END;
}
```

This macro is defined in definitions 128 and 130.
This macro is invoked in definition 124.

For each actual parameter, we check that variables are given where for `VAR` formal parameters, that IO objects are given for and only for `IO` formal parameters, and that the type of the formal and actual parameter agree: `CONST` parameters must be compatible, `VAR` and `IO` parameters must have identical type.

129 *Expression List Analysis*[129] ≡

```
{
RULE rActParam :
  ActParam IS Expr
COMPUTE
  .Key = /* formal parameter: */
    FetchKeyFromArray (ActParam.InhParams, ActParam.Paramcounter)
      DEPENDS_ON ActParam.known;
  .Type = GetType (.Key, NoKey) DEPENDS_ON ActParam.known;
  .Access = GetAccess (.Key, NoAcc) DEPENDS_ON ActParam.known;
  ActParam.MayBeWritten =
    IF (IsVarAcc (.Access), Expr.MayBeUsed, NoDefTblKeySet);
  ActParam.MayBeRead = DSunite (Expr.MayBeRead, Expr.MayBeUsed);
  ActParam.Paramcounter = ADD (ActParam.Paramcounter, 1);
```

```
  ActParam.known = ORDER (
    IF (AND (NOT (IsIoAcc (.Access)), IsIoAcc (Expr.Access)),
      Message (ERROR, "IO object not allowed for this parameter")),
    IF (AND (AND (IsVarAcc (.Access),
        NOT (IsVarAcc (Expr.Access))), NE (Expr.Access, NoAcc)),
      Message (ERROR, "Variable needed as parameter")),
    IF (AND (AND (IsIoAcc (.Access),
        NOT (IsIoAcc (Expr.Access))), NE (Expr.Access, NoAcc)),
      Message (ERROR, "IO object needed as parameter")),
    IF (/* Different types for VAR or IO parameter and argument: */
        AND (AND (AND (NOT (IsConstAcc (.Access)),
        NE (.Type, NoKey)), NE (Expr.Type, NoKey)), NE (.Type, Expr.Type)),
      Message2 (ERROR, "Argument has wrong type: %s (expected: %s)",
                SymString (GetSym (Expr.Type, NoSym)),
                SymString (GetSym (.Type, NoSym)))),
    IF (/* Incompatible types for CONST parameter and argument: */
        AND (AND (AND (IsConstAcc (.Access),
        NE (.Type, NoKey)), NE (Expr.Type, NoKey)),
        NOT (OilIsValidOp (OilIdOp2 (AssignOp, DefTbl2Oil (.Type),
                                     DefTbl2Oil(Expr.Type))))),
      Message2 (ERROR, "Argument has incompatible type: %s (expected: %s)",
                SymString (GetSym (Expr.Type, NoSym)),
                SymString (GetSym (.Type, NoSym)))));
END;

RULE rArgumentList :
  ArgumentList ::= ExprList
COMPUTE
  TRANSFER InhParams;
  ArgumentList.MayBeRead = ExprList CONSTITUENTS ActParam.MayBeRead
                           WITH (DefTblKeySet, DSunite, IDENTICAL, DSempty);
  ArgumentList.MayBeWritten = ExprList CONSTITUENTS ActParam.MayBeWritten
                              WITH (DefTblKeySet, DSunite, IDENTICAL, DSempty);
END;

RULE rArgumentList0 :
  ArgumentList ::=
COMPUTE
  ArgumentList.MayBeRead = NoDefTblKeySet;
  ArgumentList.MayBeWritten = NoDefTblKeySet;
END;

RULE rExprList1 :
  ExprList ::=  ActParam
COMPUTE
  TRANSFER InhParams;
END;

RULE rExprList :
  ExprList ::=  ExprList ',' ActParam
COMPUTE
  TRANSFER InhParams;
END;
}
```

This macro is invoked in definition 144.

The analysis of object procedure calls is analogous, except that we have to determine the procedure key from the object context. If the object is a `ParVariableK` then the call is a parallel call, otherwise it is a sequential call.

130 *Procedure Call Analysis*[130] ≡

```
{
 RULE rObjectProcedureCall :
   ObjectProcedureCall ::=  Object '.' ObjectProcedureId '(' ArgumentList ')'
 COMPUTE
   CHAINSTART ArgumentList.Paramcounter = 0;
   .Kind = GetKind (Object.Type, UndefinedK)
           DEPENDS_ON ObjectProcedureCall.known;
   ObjectProcedureId.ScopeKey =
     IF (OR (EQ (GetKind (Object.Type, UndefinedK), NodeArrayTypeK),
         EQ (GetKind (Object.Type, UndefinedK), NodeGroupTypeK)),
       GetType (Object.Type, NoKey),
       /* else */
       Object.Type) DEPENDS_ON ObjectProcedureCall.known;
   ArgumentList.InhParams = GetParams (ObjectProcedureId.Key, NoKeyArray)
     DEPENDS_ON ObjectProcedureCall.known;
   .nr = KeyArraySize (ArgumentList.InhParams);
   ObjectProcedureCall.MayBeRead =
      DSunite (ArgumentList.MayBeRead,
               IF (EQ (Object.MayBeUsed, NoDefTblKeySet),   /* pure ME call */
                 GetMayBeRead (ObjectProcedureId.Key, NoDefTblKeySet),
                 DSunite (Object.MayBeRead, Object.MayBeUsed)));
   ObjectProcedureCall.MayBeWritten =
     IF (EQ (Object.MayBeUsed, NoDefTblKeySet),  /* pure ME call */
       DSunite (GetMayBeWritten (ObjectProcedureId.Key, NoDefTblKeySet),
                ArgumentList.MayBeWritten),
       Object.MayBeUsed /* ??? */);
   ObjectProcedureCall.known = ORDER (
     IF (AND (EQ (ObjectProcedureId.Kind, ProcedureK),
         NE (.nr, ArgumentList.Paramcounter)),
       Message2 (ERROR, "Object procedure call has %d parameters, expected: %d",
                 ArgumentList.Paramcounter, .nr)),
     IF (EQ (Object.Kind, ParVariableSelK),
       Message (ERROR, "Calls are impossible for parallel variable selections"),
     /* else */
     IF (AND (AND (NE (Object.Kind, ParVariableK),
         NE (Object.Type, Subroutine ME Type[109])), NE (Object.Type, NoKey)),
       Message (ERROR, "Use '[]': Call needs parallel variable"))))
     DEPENDS_ON (.done1);
   .IsUsed =
     IF (EQ (Object.Kind, ParVariableK), /* call introducing parallelism: */
       IF (AND (EQ (.Kind, ConTypeK),
                NE (Object.Mode, IF (conAtOut, OutMode, InMode)/*!!!*/)),
         usedAR, /*else*/ usedA),
     /*else*/                            /* call on same level: */
       used) DEPENDS_ON ObjectProcedureCall.known;
   .done1 =
     SetIsUsed (ObjectProcedureId.Key, .IsUsed,
                BITOR (GetIsUsed (ObjectProcedureId.Key, used0), .IsUsed));
 END;

 SYMBOL ObjectProcedureId INHERITS FieldUse, NoKeyMsg, NameOccurrence END;
```

```
RULE rObjectProcedureId:
  ObjectProcedureId ::=  LowercaseIdent
COMPUTE
  ObjectProcedureId.Kind = GetKind (ObjectProcedureId.Key, UndefinedK)
    DEPENDS_ON ObjectProcedureId.known;
  ObjectProcedureId.known =
    IF (EQ (ObjectProcedureId.Kind, FunctionK),
      Message (ERROR, "Result of object function call not used"),
    /* else */
    IF (NE (GetKind (ObjectProcedureId.Key, ProcedureK), ProcedureK),
      Message2 (ERROR, "%s is not a PROCEDURE in type %s",
                SymString (ObjectProcedureId.Sym),
                SymString (GetSym (ObjectProcedureId.ScopeKey, NoSym)))));
END;
}
```

This macro is defined in definitions 128 and 130.
This macro is invoked in definition 124.

`MultiObjectProcedureCall`s are not different from pure sequencing as far as semantic checking is concerned and thus do not need any computations here.

21.4 Reduction statement

For a reduction statement we have to check (1) that the object given is a parallel variable selection; (2) that the context is that of a node subroutine (for connection reduction) or network subroutine (for node reduction), or global subroutine (for network reduction); (3) that the function identifier given is really that of a reduction function; (4) that the reduction function type and reduction object type are assignment compatible; (5) that the type of the target object and the result type of the reduction function are assignment compatible; and (6) that the destination object is a variable. The type of the reduced objects is marked to need a general `a_REDUCTION` procedure.

Reduction Statement Analysis[131] ≡ 131

```
{
RULE rReductionStmt :
  ReductionStmt ::=  'REDUCTION' Object ':' ReductionFunctionId 'INTO' Object
COMPUTE
  .Type  = Object[1].Type DEPENDS_ON ReductionStmt.known;
  .Type2 = Object[2].Type DEPENDS_ON ReductionStmt.known;
  .Kind  = GetKind (Object[1].ParVarType, UndefinedK)
           DEPENDS_ON ReductionStmt.known;
  .Kind2 = GetKind (Subroutine ME Type[109], UndefinedK)
           DEPENDS_ON ReductionStmt.known;
  ReductionStmt.known =
    IF (EQ (Context Kind[111], GlobalSubroutineContext),
      SetNeedNetReduction (Object[1].Type, true, true),
    /* else */
    IF (EQ (.Kind2, NetTypeK),
      SetNeedNodeReduction (Object[1].Type, true, true),
    /* else */
    IF (EQ (.Kind2, NodeTypeK),
      SetNeedConReduction (Object[1].Type, true, true))))
    DEPENDS_ON (TAIL.known, .done1, .done2);
  ReductionStmt.MayBeRead =
    DSunite (DSunite (Object[1].MayBeRead, Object[1].MayBeUsed),
             Object[2].MayBeRead);
```

```
  ReductionStmt.MayBeWritten = Object[2].MayBeUsed;
  /* (1): */
  .done1 = ORDER (
    IF (NE (Object[1].Kind, ParVariableSelK),
      Message (ERROR, "REDUCTION works only for parallel variable selection")),
    /* (2): */
    IF (AND (OR (OR (EQ (.Kind, NodeGroupTypeK), EQ (.Kind, NodeArrayTypeK)),
           EQ (.Kind, NodeTypeK)),
        NE (GetKind (Subroutine ME Type[109], NetTypeK), NetTypeK)),
    Message (ERROR, "REDUCTION <nodes> is allowed in NETWORK subroutines only")),
    IF (AND (EQ (.Kind, ConTypeK),
        NE (GetKind (Subroutine ME Type[109], NodeTypeK), NodeTypeK)),
   Message (ERROR, "REDUCTION <connections> allowed in NODE subroutines only")));
  /* (4) object compatible to reduction function ? : */
  .done2 =
    IF (AND (AND (NE (ReductionFunctionId.Type, NoKey), NE (.Type, NoKey)),
        NE (.Type, ReductionFunctionId.Type)),
      Message3 (ERROR, "Type conflict: REDUCTION %s:%s(%s)",
                SymString (GetSym (.Type, NoSym)),
                SymString (ReductionFunctionId.Sym),
                SymString (GetSym (ReductionFunctionId.Type, NoSym))),
    /* (5) reduction function compatible to target object ? : */
    IF (AND (AND (NE (ReductionFunctionId.Type, NoKey), NE (.Type2, NoKey)),
        NE (ReductionFunctionId.Type, .Type2)),
      Message2 (ERROR, "Type conflict: REDUCTION %s INTO %s",
                SymString (GetSym (ReductionFunctionId.Type, NoSym)),
                SymString (GetSym (.Type2, NoSym))),
    IF (NOT (IsVarAcc (Object[2].Access)),
      Message (ERROR, "REDUCTION INTO non-VARiable"))));
END;

RULE rReductionFunctionId :
  ReductionFunctionId ::=  LowercaseIdent
COMPUTE
  .Key = ReductionFunctionId.Key;
  ReductionFunctionId.Type = GetType (.Key, NoKey)
    DEPENDS_ON ReductionFunctionId.known;
  /* (3): */
  ReductionFunctionId.known =
    IF (NE (GetKind (.Key, ReductionFunctionK), ReductionFunctionK),
      Message1 (ERROR, "%s is no REDUCTION function",
                SymString (GetSym (.Key, NoSym))))
    DEPENDS_ON ReductionFunctionId.known;
END;
}
```

This macro is invoked in definition 124.

21.5 Winner-takes-all statement

The analysis of a winner-takes-all statement consists of the following parts: (1) compute and check the selection of `Elementname` from `Object`, (2) check that the given WTA function identifier is the name of a WTA function, (3) check that the type of the element and the WTA function are compatible, (4) check that the object procedure call is valid, and (5) mark the type for which WTA is computed as needing a general WTA procedure.

Winner-takes-all Analysis[132] ≡ 132

```
{
 RULE rWtaStmt :
   WtaStmt ::=  'WTA' Object ':' Elementname '.' WtaFunctionId ':'
                ObjectProcedureId '(' ArgumentList ')'
 COMPUTE
   .Kind  = GetKind (Object[1].ParVarType, UndefinedK) DEPENDS_ON WtaStmt.known;
   .Kind2 = GetKind (Subroutine ME Type[109], UndefinedK) DEPENDS_ON WtaStmt.known;
   WtaStmt.MayBeRead =
     DSunite (ArgumentList.MayBeRead, Object[1].MayBeRead);
   WtaStmt.MayBeWritten = Object[1].MayBeUsed;
   /* (1): Selection */
   .Type =
     IF (EQ (Object.Kind, ParVariableK),
       Object.Type,
       ORDER (
         Message1 (ERROR, "WTA object must be parallel variable (has type %s)",
                   SymString (GetSym (Object.Type, NoSym))),
         NoKey)) DEPENDS_ON WtaStmt.known;
   Elementname.ScopeKey = .Type;
   Messag3 (NOTE, "%s %s from type %s",
            SymString (GetSym (Elementname.Type, NoSym)),
            SymString (Elementname.Sym),
            SymString (GetSym (Elementname.ScopeKey, NoSym)));

   /* (3): Element type compatible to wta function ? : */
   .Type2 = Elementname.Type;
   .done1 =
     IF (AND (AND (NE (WtaFunctionId.Type, NoKey), NE (.Type2, NoKey)),
         NOT (OilIsValidOp (OilIdOp2 (AssignOp,
              DefTbl2Oil (WtaFunctionId.Type), DefTbl2Oil(.Type2))))),
       Message3 (ERROR, "Type conflict: WTA :%s.%s(%s):",
                 SymString (GetSym (.Type2, NoSym)),
                 SymString (WtaFunctionId.Sym),
                 SymString (GetSym (WtaFunctionId.Type, NoSym))));

   /* (4): Object procedure call: */
   CHAINSTART ArgumentList.Paramcounter = 0;
   ObjectProcedureId.ScopeKey =
     IF (OR (EQ (GetKind (Object.Type, UndefinedK), NodeArrayTypeK),
         EQ (GetKind (Object.Type, UndefinedK), NodeGroupTypeK)),
       GetType (Object.Type, NoKey),
       /* else */
       Object.Type) DEPENDS_ON WtaStmt.known;
   ArgumentList.InhParams = GetParams (ObjectProcedureId.Key, NoKeyArray)
     DEPENDS_ON WtaStmt.known;
   .nr = KeyArraySize (ArgumentList.InhParams);
   WtaStmt.known =
     IF (AND (EQ (ObjectProcedureId.Kind, ProcedureK),
         NE (.nr, ArgumentList.Paramcounter)),
       Message2 (ERROR, "WTA procedure call has %d parameters, expected: %d",
                 ArgumentList.Paramcounter, .nr))
     DEPENDS_ON (.done1, .done2);
   /* (5): mark "need a_WTA" */
   .done2 =
```

```
    IF (EQ (.Kind2, NodeTypeK),
      SetNeedConWta (Elementname.Type, true, true),
    /* else */
    IF (EQ (.Kind2, NetTypeK),
      SetNeedNodeWta (Elementname.Type, true, true),
    /* else */
    IF (EQ (Context Kind[111], GlobalSubroutineContext),
      SetNeedNetWta (Elementname.Type, true, true))))
END;

RULE rWtaFunctionId :
  WtaFunctionId ::=  LowercaseIdent
COMPUTE
  .Key = WtaFunctionId.Key;
  /* (2): check WTA function id */
  WtaFunctionId.Type =
    IF (NE (GetKind (.Key, UndefinedK), WtaFunctionK),
      ORDER (Message1 (ERROR, "%s is no WTA function",
                       SymString (GetSym (.Key, NoSym))),
             NoKey),
      /* else */
      GetType (WtaFunctionId.Key, NoKey))
    DEPENDS_ON WtaFunctionId.known;
  WtaFunctionId.known = WtaFunctionId.Type;
END;
}
```

This macro is invoked in definition 124.

21.6 Control flow

We will discuss the control flow statements in the following order:

133 *Control Flow Analysis*[133] ≡

{
Return Statement Analysis[134]
If Statement Analysis[135]
Loop Statement Analysis[136]
Break Statement Analysis[137]
}

This macro is invoked in definition 124.

For the `RETURN` statement, the only thing we have to check is whether the return type is correct: For procedures, the return statement without an expression has to be used (implicit `Void` type); for functions, the return statement with an expression has to be used and the type of that expression must be compatible with the function's return type. We do not currently check whether a `RETURN` statement is present at the end of a function.

134 *Return Statement Analysis*[134] ≡

```
{
RULE rReturnStmtVoid :
  ReturnStmt ::=  'RETURN'
COMPUTE
  ReturnStmt.MayBeRead = NoDefTblKeySet;
  ReturnStmt.known =
    IF (NOT (EQ (Subroutine Return Type[108], VoidKey)),
```

```
      Message (ERROR, "No function value given to be RETURNed"))
    DEPENDS_ON ReturnStmt.known;
END;

RULE rReturnStmt :
  ReturnStmt ::=  'RETURN' Expr
COMPUTE
  TRANSFER MayBeRead;
  ReturnStmt.known =
    IF (EQ (Subroutine Return Type[108], VoidKey),
      Message (ERROR, "Cannot RETURN a value from a procedure"),
    /* else */
    IF (AND (AND (NE (Subroutine Return Type[108], NoKey),
        NE (Expr.Type, NoKey)),
        NOT (OilIsValidOp (OilIdOp2 (AssignOp,
             DefTbl2Oil (Subroutine Return Type[108]),
                        DefTbl2Oil (Expr.Type))))),
      Message2 (ERROR, "wrong RETURN type: %s (expected: %s)",
                SymString (GetSym (Subroutine Return Type[108], NoSym)),
                SymString (GetSym (Expr.Type, NoSym)))))
    DEPENDS_ON ReturnStmt.known;
END;
}
```

This macro is invoked in definition 133.

The thing to check for an `IF` statement is that all conditional expressions given must have type `Bool`:

If Statement Analysis[135] ≡ 135

```
{
RULE rIfStmt :
  IfStmt ::=  'IF' Expr 'THEN' Statements ElsePart 'END' OptIF
COMPUTE
  IfStmt.MayBeRead = DSunite (Expr.MayBeRead, ElsePart.MayBeRead);
  Statements.known =
    IF (AND (NE (Expr.Type, NoKey), NE (Expr.Type, BoolKey)),
      Message1 (ERROR, "IF condition has wrong type: %s (expected: Bool)",
                SymString (GetSym (Expr.Type, NoSym))))
    DEPENDS_ON Expr.known;
END;

RULE rElsif :
  ElsePart ::=  'ELSIF' Expr 'THEN' Statements ElsePart
COMPUTE
  ElsePart[1].MayBeRead = DSunite (Expr.MayBeRead, ElsePart[2].MayBeRead);
  Statements.known =
    IF (AND (NE (Expr.Type, NoKey), NE (Expr.Type, BoolKey)),
      Message1 (ERROR, "ELSIF condition has wrong type: %s (expected: Bool)",
                SymString (GetSym (Expr.Type, NoSym))))
    DEPENDS_ON Expr.known;
END;

RULE rElse0 :
  ElsePart ::=
COMPUTE
  ElsePart.MayBeRead = NoDefTblKeySet;
END;
```

```
RULE rElse :
  ElsePart ::=  'ELSE' Statements
COMPUTE
  ElsePart.MayBeRead = NoDefTblKeySet;
END;
}
```

This macro is invoked in definition 133.

To analyze `BREAK` statements, we compute the attribute `LoopNest` for each loop and set it to zero in the root context. A break is rejected if the `LoopNest` is zero. For the normal loop (WHILE and/or UNTIL) we only have to check that the conditions given have boolean type. For FOR loops we also have to compute a value for the step and check that the loop limits are integer.

136 *Loop Statement Analysis*[136] ≡

```
{
SYMBOL CupitProgram COMPUTE
  INH.LoopNest = 0;
END;

SYMBOL Statements COMPUTE
  INH.LoopNest = INCLUDING (CupitProgram.LoopNest, Statements.LoopNest);
    /* except for LoopStmt bodies, see below */
END;

RULE rLoopStmt :
  LoopStmt ::=  OptWhilePart 'REPEAT' Statements OptUntilPart 'END' OptREPEAT
COMPUTE
  Statements.LoopNest = ADD (INCLUDING Statements.LoopNest, 1);
  LoopStmt.MayBeRead = DSunite (OptWhilePart.MayBeRead, OptUntilPart.MayBeRead);
  LoopStmt.MayBeWritten = NoDefTblKeySet;
END;

RULE rForLoopStmt :
  LoopStmt ::=  'FOR' Object ':=' Expr ForLoopStep Expr 'REPEAT'
                Statements OptUntilPart 'END' OptREPEAT
COMPUTE
  Statements.LoopNest = ADD (INCLUDING Statements.LoopNest, 1);
  LoopStmt.MayBeRead =
    DSunite (DSunite (Expr[1].MayBeRead, Expr[2].MayBeRead),
             DSunite (Object.MayBeRead, OptUntilPart.MayBeRead));
  LoopStmt.MayBeWritten = Object.MayBeUsed;
  Statements.known =
   IF (OR (NOT (IsInt (Expr[1].Type)), NOT (IsInt (Expr[2].Type))),
    Message3 (ERROR, "'FOR' loop limits have wrong types: %s/%s (expected: %s)",
              SymString (GetSym (Expr[1].Type, NoSym)),
              SymString (GetSym (Expr[2].Type, NoSym)),
              "Integers")) DEPENDS_ON LoopStmt.known;
END;

RULE rOptWhilePart :
  OptWhilePart ::=  'WHILE' Expr
COMPUTE
  OptWhilePart.MayBeRead = Expr.MayBeRead;
  OptWhilePart.known =
    IF (NE (Expr.Type, BoolKey),
```

```
      Message1 (ERROR, "WHILE condition has wrong type: %s (expected: Bool)",
                 SymString (GetSym (Expr.Type, NoSym))))
    DEPENDS_ON OptWhilePart.known;
END;

RULE rOptWhilePart0 :
  OptWhilePart ::=
COMPUTE
  OptWhilePart.MayBeRead = NoDefTblKeySet;
END;

RULE rOptUntilPart :
  OptUntilPart ::=  'UNTIL' Expr
COMPUTE
  OptUntilPart.MayBeRead = Expr.MayBeRead;
  OptUntilPart.known =
    IF (NE (Expr.Type, BoolKey),
      Message1 (ERROR, "UNTIL condition has wrong type: %s (expected: Bool)",
                 SymString (GetSym (Expr.Type, NoSym))))
    DEPENDS_ON OptUntilPart.known;
END;

RULE rOptUntilPart0 :
  OptUntilPart ::=
COMPUTE
  OptUntilPart.MayBeRead = NoDefTblKeySet;
END;

RULE rUpto :   ForLoopStep ::=  'UPTO'   COMPUTE  ForLoopStep.nr =  1; END;
RULE rTo :     ForLoopStep ::=  'TO'     COMPUTE  ForLoopStep.nr =  1; END;
RULE rDownto : ForLoopStep ::=  'DOWNTO' COMPUTE  ForLoopStep.nr = NEG(1); END;
}
```

This macro is invoked in definition 133.

Break Statement Analysis[137] ≡ 137

```
{
RULE rBreakStmt :
  BreakStmt ::=  'BREAK'
COMPUTE
  BreakStmt.known =
    IF (LT (INCLUDING Statements.LoopNest, 1),
      Message (ERROR, "This BREAK appears outside of any loop"))
    DEPENDS_ON BreakStmt.known;
END;
}
```

This macro is invoked in definition 133.

21.7 Data allocation

The data allocation and deallocation statements are all applicable in certain contexts only. The context also determines what type the object(s) to work with must have.

For the `REPLICATE` statement the object must be a network when the statement appears in the central agent, or it must be a node or connection when the statement appears in an object procedure of the object's type. In all other contexts, `REPLICATE` is not allowed. The expression can be an integer (for networks, nodes, connections) or integer interval (for networks only).

138 *Data allocation Analysis*[138] ≡

```
{
 RULE rReplicateInto :
   DataAllocationStmt ::=  'REPLICATE' Object 'INTO' Expr
 COMPUTE
   .Kind = GetKind (Object.Type, UndefinedK)
     DEPENDS_ON DataAllocationStmt.known;
   .Sym  = GetSym (Object.Type, NoSym)
     DEPENDS_ON DataAllocationStmt.known;
   DataAllocationStmt.MayBeRead = Expr.MayBeRead;
   DataAllocationStmt.MayBeWritten = NoDefTblKeySet;
   .done1 =
     IF (NOT (OR (OR (EQ (Expr.Type, NoKey), IsInt (Expr.Type)),
              AND (EQ (.Kind, NetTypeK), IsInterval (Expr.Type)))),
         Message1 (ERROR, "INTO expression must have integral type (is: %s)",
                   SymString (GetSym (Expr.Type, NoSym))));
   DataAllocationStmt.known =
     IF (EQ (.Kind, NetTypeK),
       IF (NE (Context Kind[111], GlobalSubroutineContext),
         Message1 (ERROR, "REPLICATE %s is allowed in the central agent only",
                   SymString (.Sym))),
     /* else */
     IF (OR (EQ (.Kind, ConTypeK), EQ (.Kind, NodeTypeK)),
       IF (NE (Subroutine ME Type[109], Object.Type),
         Message2 (ERROR, "REPLICATE %s is allowed in %s subroutines only",
                   SymString (.Sym), SymString (.Sym))),
     /* else */
     IF (NOT (EQ (Object.Type, NoKey)),
       Message1 (ERROR, "REPLICATE is not allowed for objects of type %s",
                 SymString (.Sym)))))
     DEPENDS_ON (TAIL.known, .done1);
 END;
}
```

This macro is defined in definitions 138, 139, 140, and 141.
This macro is invoked in definition 124.

For the `EXTEND` statement the object must be a node group and the statement must appear in a network subroutine; everything else is illegal. The expression must be an integer.

139 *Data allocation Analysis*[139] ≡

```
{
 RULE rExtendBy :
   DataAllocationStmt ::=  'EXTEND' Object 'BY' Expr
 COMPUTE
   .Kind = GetKind (Object.Type, UndefinedK)
     DEPENDS_ON DataAllocationStmt.known;
   .Sym  = GetSym (Object.Type, NoSym)
     DEPENDS_ON DataAllocationStmt.known;
   DataAllocationStmt.MayBeRead = Expr.MayBeRead;
   DataAllocationStmt.MayBeWritten = NoDefTblKeySet;
   .done1 =
     IF (NOT (OR (EQ (Expr.Type, NoKey), IsInt (Expr.Type))),
       Message1 (ERROR, "BY expression must have integral type (is: %s)",
                 SymString (GetSym (Expr.Type, NoSym))))
     DEPENDS_ON DataAllocationStmt.known;
   DataAllocationStmt.known =
     IF (EQ (.Kind, NodeGroupTypeK),
```

```
        IF (OR (NE (Context Kind[111], ObjSubroutineContext),
               NE (GetKind (Subroutine ME Type[109], NetTypeK), NetTypeK)),
             Message (ERROR, "EXTEND is allowed in network subroutines only"),
             /* else */
             Messag1 (NOTE, "EXTEND %s", SymString (.Sym))),
        /* else */
        IF (EQ (.Kind, NodeArrayTypeK),
          Message (ERROR, "You can EXTEND only GROUPs, not ARRAYs"),
        /* else */
        IF (NOT (EQ (Object.Type, NoKey)),
          Message1 (ERROR, "EXTEND is not allowed for objects of type %s",
                    SymString (.Sym)))))
        DEPENDS_ON (TAIL.known, .done1);
    END;
    }
```

This macro is defined in definitions 138, 139, 140, and 141.
This macro is invoked in definition 124.

The `CONNECT` and `DISCONNECT` statements are allowed in network subroutines only. The objects given must be parallel variable selections of connection type.

Data allocation Analysis[140] ≡ 140

```
  {
    RULE rConnectTo :
      DataAllocationStmt ::=  'CONNECT' Object 'TO' Object
    COMPUTE
      CONNECT object analysis[142]('CONNECT')
    END;
  }
```

This macro is defined in definitions 138, 139, 140, and 141.
This macro is invoked in definition 124.

Data allocation Analysis[141] ≡ 141

```
  {
    RULE rDisconnectFrom :
      DataAllocationStmt ::=  'DISCONNECT' Object 'FROM' Object
    COMPUTE
      CONNECT object analysis[142]('DISCONNECT')
    END;
  }
```

This macro is defined in definitions 138, 139, 140, and 141.
This macro is invoked in definition 124.

Since the tests for `CONNECT` and `DISCONNECT` statements are the same, they are collected in this extra FunnelWeb macro. The statement type is a parameter which is used in the text of the error messages.

CONNECT object analysis[142](◇1) ≡ 142

```
  {
      DataAllocationStmt.MayBeRead =
        DSunite (Object[1].MayBeRead, Object[2].MayBeRead);
      DataAllocationStmt.MayBeWritten =
        DSunite (Object[1].MayBeUsed, Object[2].MayBeUsed);
      DataAllocationStmt.known =
        IF (OR (EQ (Object[1].Type, NoKey), EQ (Object[2].Type, NoKey)),
          0,  /* do nothing, a message will be generated elsewhere */
        /* else */
        IF (OR (EQ (Object[1].Kind, ParVariableK),
                EQ (Object[2].Kind, ParVariableK)),
```

```
      Message (ERROR, "Cannot ◇1 parallel variables"),
    /* else */
    IF (OR (NE (GetKind (Object[1].Type, UndefinedK), ConTypeK),
            NE (GetKind (Object[2].Type, UndefinedK), ConTypeK)),
      Message (ERROR, "Objects at ◇1 must be connection interfaces"),
    /* else */
    ORDER (
      IF (NE (Object[1].Mode, OutMode),
        Message (ERROR, "Left object must be OUT interface")),
      IF (NE (Object[2].Mode, InMode),
        Message (ERROR, "Right object must be IN interface")),
      IF (NE (Object[1].Type, Object[2].Type),
          ORDER (Message2 (ERROR, "Connection type conflict: %s/%s",
                           SymString (GetSym (Object[1].Type, NoSym)),
                           SymString (GetSym (Object[2].Type, NoSym))),
                 NoKey))))))
    DEPENDS_ON TAIL.known;
}
```

This macro is invoked in definitions 140 and 141.

21.8 Merge statement

An explicit `MERGE` statement is allowed only in global subroutines and can be applied only to `NETWORK` objects.

143 *Merge Statement Analysis*[143] ≡

```
{
RULE rMergeStmt:
  MergeStmt ::=  'MERGE' Object
COMPUTE
  MergeStmt.nr = 1; /* indicate presence of MERGE */
  MergeStmt.known =
    ORDER (
      IF (NE (Context Kind[111], GlobalSubroutineContext),
        Message (ERROR, "MERGE is allowed in the central agent only")),
      IF (NE (GetKind (Object.Type, NetTypeK), NetTypeK),
        Message (ERROR, "MERGE can be applied to NETWORK objects only")))
    DEPENDS_ON Object.known;
END;
}
```

This macro is invoked in definition 124.

22 Expressions

144 *Expression Analysis*[144] ≡

{
Expression List Analysis[129]
Ternary Expression Analysis[170]
Binary Expression Analysis[171]
Unary Expression Analysis[174]
Type Conversion Expression Analysis[176]
Denoter Expression Analysis[178]
Function Call Analysis[182]

Object Expression Analysis[183]

```
SYMBOL Expr COMPUTE
  SYNT.Operator = OilErrorOp();
  SYNT.MayBeUsed = NoDefTblKeySet;
END;
}
```

This macro is invoked in definition 193.

There are several important attributes that an expression may have: First of all, each expression has a unique type. This type is always determined independent of the expression's context in CuPit.
Second, an expression may be constant. The values of constant expressions are computed at compile time ("constant folding"). A constant expression can be recognized by the value `ConstantK` in the `Kind` attribute (and always has `ConstAcc` in the `Access` attribute). Non-constant expressions have kind `VariableK`.
Third, it may be allowed or not allowed to assign to an expression. This property is important to decide whether an expression may be used as an argument for a `VAR` or `IO` formal parameter. Assignable expressions have an `Access` attribute value of `VarAcc` or `VarPAcc` or of `IoAcc` or `IoPAcc` and kind `VariableK`.

For expressions consisting of an operator application to one, two, or three other expressions, the `Operator` attribute contains the exact Oil identification of the operator used. Since this attribute is not valid for other types of expressions, we have to set a default. This is done in the symbol attribution for `Expr` above.

22.1 Operator identification

To simplify the semantic checking of expressions, we use Eli's Operator Identification Language (OIL). Although we generate C code, where we could leave many operators unidentified we want to identify operators in order to be able to perform constant folding. In addition, using Oil saves a lot of work for performing the type checking of operands.

Since we do not use the Oil representation of types all the time (because we have to store properties of types, too), we have to define conversion routines between the oil type representation and our definition-table-based type representation. We define one routine `DefTbl2Oil` to generate oil type identifiers from definition table keys that identify types and a second routine `Oil2DefTbl` to generate definition table keys from oil type identifiers. To announce such a correspondence, `DefineOilType` must be used.

oil_interface.h[145] ≡ 145

```
{
#ifndef oil_interface_H
#define oil_interface_H

#include "deftbl.h"
#include "oiladt2.h"
#include "pdl_gen.h"

void         DefineOilType (DefTableKey d, tOilType o);
tOilType     NewSymbolicOilType (DefTableKey d);
tOilType     DefTbl2Oil (DefTableKey d);
DefTableKey  Oil2DefTbl (tOilType o);

#define IsInt(T)         (OR (OR (EQ (T, Int1Key), EQ (T, Int2Key)), \
                          EQ (T, IntKey)))
#define IsInterval(T)    (OR (OR (EQ (T, Interval1Key), EQ (T, Interval2Key)), \
                          EQ (T, IntervalKey)))
```

```
#endif
}
```

This macro is attached to an output file.

146 **oil_interface.c**[146] ≡

```
{
#include "cupit.h"
#include "scope.h"
#include "type.h"
#include "oil_interface.h"
#include "OilDecls.h"

Define Oil Type[147]
New Symbolic Type In Oil[149]
Definition Table To Oil[150]
Oil To Definition Table[151]
}
```

This macro is attached to an output file.

When a corresponding pair of a definition table key (representing a type name) and a `tOilType` is given, we store the oil type as a property in the definition table and the definition table key as an entry in an array `OilTypeKeys` indexed by the oil types. The latter is possible only since OIL guarantees that the `OilTypeName` function returns minimal unique integers for the set of defined Oil types.

147 *Define Oil Type*[147] ≡

```
{
#define MaxNrOfTypes 1000
static DefTableKey OilTypeKeys[MaxNrOfTypes];

void DefineOilType (DefTableKey d, tOilType o)
{
  SetOilType (d, o, o);
  OilTypeKeys[OilTypeName(o)] = d;
}
}
```

This macro is invoked in definition 146.

The standard types have to be announced using this mechanism.

148 *Set Oil Types For Standard Types*[148] ≡

```
{
DefineOilType (BoolKey, oilBool);
DefineOilType (IntKey,  oilInt);
DefineOilType (Int2Key, oilInt2);
DefineOilType (Int1Key, oilInt1);
DefineOilType (RealKey, oilReal);
DefineOilType (StringKey, oilString);
DefineOilType (IntervalKey,  oilInterval);
DefineOilType (Interval2Key, oilInterval2);
DefineOilType (Interval1Key, oilInterval1);
DefineOilType (RealervalKey, oilRealerval);
}
```

This macro is invoked in definition 88.

Whenever a new symbolic type is introduced, we must instantiate the Oil symbolic class, and announce and return the resulting Oil type indentifier:

149 *New Symbolic Type In Oil*[149] ≡
```
{
tOilType NewSymbolicOilType (DefTableKey d)
{
  tOilType result = OilClassInst0 (SymbolicClass, SymbolicClass_name);
  DefineOilType (d, result);
  return (result);
}
}
```
This macro is invoked in definition 146.

The construction of Oil type identifiers from type definition table keys works by simply recalling the appropriate property value.

Definition Table To Oil[150] ≡ 150
```
{
tOilType DefTbl2Oil (DefTableKey d)
{
  return (GetOilType (d, OilErrorType()));
}
}
```
This macro is invoked in definition 146.

The backward conversion from Oil type identifiers to definition table keys is performed by recalling the appropriate array entry.

Oil To Definition Table[151] ≡ 151
```
{
DefTableKey Oil2DefTbl (tOilType o)
{
  int nr = OilTypeName(o);
  return (nr < 0 ? NoKey : OilTypeKeys[nr]);
}
}
```
This macro is invoked in definition 146.

Given these converion routines, we describe the available operators in Oil:

types.oil[152] ≡ 152

{
Oil Coercions[153]
Oil Operators[154]
Oil Symbolic Class[166]
Oil Structure Class[167]
}

This macro is attached to an output file.

The set of coercions (implicit type conversions) is very small: The only ones allowed are the promotion of a small integer type to the next larger integer type.

Oil Coercions[153] ≡ 153
```
{
COERCION mkInt2 (oilInt1) : oilInt2;
COERCION mkInt  (oilInt2) : oilInt;
}
```
This macro is invoked in definition 152.

To describe the set of available operators, we first define a number of sets of oil types. We do never discriminate the integer types.

Oil Operators[154] ≡ 154
```
{
SET Ints    = [oilInt, oilInt1, oilInt2];
SET Ariths  = Ints + [oilReal];
SET Simples = Ariths + [oilBool, oilString];
SET Intervals = [oilInterval, oilInterval1, oilInterval2];
SET Builtins = Simples + Intervals;
}
```
This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The ternary operator is defined for all builtin types if the second and third operand are compatible. (The first operand is tested “by hand”).

155 *Oil Operators*[155] ≡
```
{
INDICATION TernOp:   aTernOp;
OPER aTernOp (Builtins, Builtins) : Builtins;
}
```
This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The boolean operators are defined for boolean operands only.

156 *Oil Operators*[156] ≡
```
{
INDICATION AndOp: bAndOp;
INDICATION OrOp:  bOrOp;
INDICATION XorOp: bXorOp;
OPER bAndOp, bOrOp, bXorOp (oilBool, oilBool) : oilBool;
}
```
This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The tests for equality and inequality are defined for all number and interval types, for symbolic types, and for `String`. They are, however, not defined for `Bool`. The operators with different operand types have to be discriminated, though, since the constant folding operations for them are different and for some variants different code has to be generated.

157 *Oil Operators*[157] ≡
```
{
INDICATION EqOp:     iEqOp, rEqOp, IntervalEqOp, RealervalEqOp,
                     StringEqOp, SymbolicEqOp;
INDICATION NeqOp:    iNeqOp, rNeqOp, IntervalNeqOp, RealervalNeqOp,
                     StringNeqOp, SymbolicNeqOp;
OPER iEqOp, iNeqOp (Ints, Ints) : oilBool;
OPER rEqOp, rNeqOp (oilReal, oilReal) : oilBool;
OPER IntervalEqOp, IntervalNeqOp (Intervals, Intervals) : oilBool;
OPER RealervalEqOp, RealervalNeqOp (oilRealerval, oilRealerval) : oilBool;
OPER StringEqOp, StringNeqOp (oilString, oilString) : oilBool;
}
```
This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The comparison operators for greater or less are defined for number types and interval types. Again, discrimination is necessary to enable proper constant folding and code generation.

158 *Oil Operators*[158] ≡
```
{
```

```
INDICATION LtOp:     iLtOp, rLtOp;
INDICATION GtOp:     iGtOp, rGtOp;
INDICATION LeOp:     iLeOp, rLeOp;
INDICATION GeOp:     iGeOp, rGeOp;
OPER iLtOp, iGtOp, iLeOp, iGeOp (Ints, Ints) : oilBool;
OPER rLtOp, rGtOp, rLeOp, rGeOp (oilReal, oilReal) : oilBool;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The operator `IN` has mixed operand types. It is defined in three integer and one real variant, that have to be discriminated for constant folding and code generation.

Oil Operators[159] ≡ 159

```
{
INDICATION InOp:     IntInOp, IntInOp1, IntInOp2, RealInOp;
OPER IntInOp  (oilInt,  Intervals) : oilBool;
OPER IntInOp1 (oilInt1, Intervals) : oilBool;
OPER IntInOp2 (oilInt2, Intervals) : oilBool;
OPER RealInOp (oilReal, oilRealerval) : oilBool;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The interval construction operator is available once for each basic number type:

Oil Operators[160] ≡ 160

```
{
INDICATION IntervalOp:     IntIntervalOp, IntIntervalOp1,
                           IntIntervalOp2, RealIntervalOp;
OPER IntIntervalOp  (oilInt, oilInt): oilInterval;
OPER IntIntervalOp1 (oilInt1, oilInt1) : oilInterval1;
OPER IntIntervalOp2 (oilInt2, oilInt2) : oilInterval2;
OPER RealIntervalOp (oilReal, oilReal) : oilRealerval;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The bit operators are defined for the integer types only.

Oil Operators[161] ≡ 161

```
{
INDICATION BitandOp: iBitandOp;
INDICATION BitorOp:  iBitorOp;
INDICATION BitxorOp: iBitxorOp;
INDICATION LshiftOp: iLshiftOp;
INDICATION RshiftOp: iRshiftOp;
OPER iBitandOp, iBitorOp, iBitxorOp, iLshiftOp, iRshiftOp (Ints, Ints) : Ints;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The five basic arithmetic operators are defined for the integer and real types. Two variants have to be discriminated for proper constant folding.

Oil Operators[162] ≡ 162

```
{
INDICATION PlusOp:   iPlusOp, rPlusOp;
INDICATION MinusOp:  iMinusOp, rMinusOp;
```

```
INDICATION MulOp:    iMulOp, rMulOp;
INDICATION DivOp:    iDivOp, rDivOp;
INDICATION ModOp:    iModOp, rModOp;
OPER iPlusOp, iMinusOp, iMulOp, iDivOp, iModOp (Ints, Ints) : Ints;
OPER rPlusOp, rMinusOp, rMulOp, rDivOp, rModOp (oilReal, oilReal) : oilReal;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

The exponentiation operator comes in exactly three flavors. Note that it is not defined separately for each integer type.

163 *Oil Operators*[163] ≡

```
{
INDICATION ExpOp:    iExpOp, rExpOp, riExpOp;
OPER iExpOp (oilInt, oilInt) : oilInt;
OPER rExpOp (oilReal, oilReal) : oilReal;
OPER riExpOp (oilReal, oilInt) : oilReal;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

Negation is defined for integer and real, Not is defined for Bool only, bitwise not is defined for integer, Min, Max and Random are defined for each interval type individually.

164 *Oil Operators*[164] ≡

```
{
INDICATION NegOp:    iNegOp, rNegOp;
INDICATION NotOp:    bNotOp;
INDICATION BitnotOp: iBitnotOp;
INDICATION MinOp:    iMinOp, iMinOp1, iMinOp2, rMinOp;
INDICATION MaxOp:    iMaxOp, iMaxOp1, iMaxOp2, rMaxOp;
INDICATION RandomOp: iRandomOp, iRandomOp1, iRandomOp2, rRandomOp;
OPER iNegOp (Ints) : Ints;
OPER rNegOp (oilReal) : oilReal;
OPER bNotOp (oilBool) : oilBool;
OPER iBitnotOp (Ints) : Ints;
OPER iMinOp (oilInterval) : oilInt;
OPER iMinOp1 (oilInterval1) : oilInt1;
OPER iMinOp2 (oilInterval2) : oilInt2;
OPER rMinOp (oilRealerval) : oilReal;
OPER iMaxOp (oilInterval) : oilInt;
OPER iMaxOp1 (oilInterval1) : oilInt1;
OPER iMaxOp2 (oilInterval2) : oilInt2;
OPER rMaxOp (oilRealerval) : oilReal;
OPER iRandomOp (oilInterval) : oilInt;
OPER iRandomOp1 (oilInterval1) : oilInt1;
OPER iRandomOp2 (oilInterval2) : oilInt2;
OPER rRandomOp (oilRealerval) : oilReal;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

Assignment, including assignmen individualt with arithmetic, is also treated as an operator so that it can be type-checked by Oil.

165 *Oil Operators*[165] ≡

```
{
```

```
INDICATION AssignOp: ArithAssign, IntervalAssign, BoolAssign,
                     StringAssign, SymbolicAssignOp, StructureAssignOp;
INDICATION PlusAssignOp:  aPlusAssignOp;
INDICATION MinusAssignOp: aMinusAssignOp;
INDICATION MulAssignOp:   aMulAssignOp;
INDICATION DivAssignOp:   aDivAssignOp;
INDICATION ModAssignOp:   aModAssignOp;
OPER ArithAssign (Ariths, Ariths) : oilVoid;
OPER IntervalAssign (Intervals, Intervals) : oilVoid;
OPER BoolAssign (oilBool, oilBool) : oilVoid;
OPER StringAssign (oilString, oilString) : oilVoid;
OPER aPlusAssignOp (Ariths, Ariths) : oilVoid;
OPER aMinusAssignOp (Ariths, Ariths) : oilVoid;
OPER aMulAssignOp (Ariths, Ariths) : oilVoid;
OPER aDivAssignOp (Ariths, Ariths) : oilVoid;
OPER aModAssignOp (Ariths, Ariths) : oilVoid;
}
```

This macro is defined in definitions 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, and 165.
This macro is invoked in definition 152.

For each symbolic type we also define its own equality and non-equality comparison and an assignment operator:

Oil Symbolic Class[166] ≡ 166

```
{
CLASS SymbolicClass () BEGIN
  OPER SymbolicEqOp (SymbolicClass, SymbolicClass) : oilBool;
  OPER SymbolicNeqOp (SymbolicClass, SymbolicClass) : oilBool;
  OPER SymbolicAssignOp (SymbolicClass, SymbolicClass) : oilVoid;
END;
}
```

This macro is invoked in definition 152.

For structured types, only assignment is defined:

Oil Structure Class[167] ≡ 167

```
{
CLASS StructureClass () BEGIN
  OPER StructureAssignOp (StructureClass, StructureClass) : oilVoid;
END;
}
```

This macro is invoked in definition 152.

22.2 Constant folding

The CuPit compiler performs complete constant folding and global constant propagation on boolean, integer, floating point, and interval values. For this purpose, a type is needed to express the constant values. This type must be able to represent one or two integers or one or two real values. The `i1` component of this type `CupitConst` holds the value for the representation of `Bool, Int1, Int2`, and `Int` constants or the interval minimum for the representation of `Interval1, Interval2`, and `Interval` constants. `i2` holds the maximum in the integer interval cases.

`r1` holds the floating point value for the representation of `Real` constants or the interval minimum of `Realerval` constants. `r2` holds the maximum of `Realerval` constants.

The folding module has functions to initialize a `CupitConst` to a non-existing value (`SetNothing`), to an integer or real value (`SetIval`, `SetRval`), or to an integer or real interval value of integer or real intervals (`SetIminmax`, `SetRminmax`). Furthermore, there are functions to read an integer or real value (`GetIval`, `GetRval`), or to read the upper or lower bounds of integer or real intervals (`GetImax`, `GetImin`, `GetRmax`, `GetRmin`). An assertion fails, if the requested value is not conforming to the kind of the constant. The constant folding operations itself are implemented in a single function `ComputeConst`, which is parameterized with the operator to compute. For code generation purposes, there is also a function `Const2Str` that generates an external string representation from a `CupitConst`.

168 **folding.h**[168] ≡

```
{
 #ifndef folding_H
 #define folding_H

 #include "cupit.h"
 #include "oiladt2.h"

 /* Constant-types, Intconst is also used for Bool */
 typedef enum {
    Errorconst, Intconst, Realconst, Intervalconst, Realervalconst
 } ConstTag;

 typedef struct {
   ConstTag tag;
   Int      i1, i2;
   Real     r1, r2;
 } CupitConst;

 #define ErrorConst     SetNothing ()

 Bool       IsErrorConst (CupitConst cc);
 CupitConst SetNothing ();
 CupitConst SetIval (Int i);
 CupitConst SetIminmax (Int i1, Int i2);
 CupitConst SetRval (Real r);
 CupitConst SetRminmax (Real r1, Real r2);
 Int        GetIval (CupitConst cc);
 Int        GetImin (CupitConst cc);
 Int        GetImax (CupitConst cc);
 Real       GetRval (CupitConst cc);
 Real       GetRmin (CupitConst cc);
 Real       GetRmax (CupitConst cc);
 CupitConst ComputeConst (tOilOp op, CupitConst cc1, CupitConst cc2);
 char*      Const2Str (CupitConst cc);
 #endif
}
```

This macro is attached to an output file.

This is the implementation of the module:

169 **folding.c**[169] ≡

```
{
 #include "folding.h"
 #include <math.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include "OilDecls.h"
```

```
static set_cc_tag (CupitConst *cc, ConstTag tag)
{
  /* initialize all fields of a CupitConst and set its tag */
  cc->tag = tag;
  cc->i1 = cc->i2 = -1;
  cc->r1 = cc->r2 = -1.0;
}

Bool IsErrorConst (CupitConst cc)
{
  return (cc.tag == Errorconst);
}

CupitConst SetNothing ()
{
  CupitConst cc;
  set_cc_tag (&cc, Errorconst);
  return (cc);
}

CupitConst SetIval (Int i)
{
  CupitConst cc;
  set_cc_tag (&cc, Intconst);
  cc.i1 = i;
  return (cc);
}

CupitConst SetIminmax (Int i1, Int i2)
{
  CupitConst cc;
  set_cc_tag (&cc, Intervalconst);
  cc.i1 = i1;
  cc.i2 = i2;
  return (cc);
}

CupitConst SetRval (Real r)
{
  CupitConst cc;
  set_cc_tag (&cc, Realconst);
  cc.r1 = r;
  return (cc);
}

CupitConst SetRminmax (Real r1, Real r2)
{
  CupitConst cc;
  set_cc_tag (&cc, Realervalconst);
  cc.r1 = r1;
  cc.r2 = r2;
  return (cc);
}

Int GetIval (CupitConst cc)
```

```
{
  _assert (cc.tag == Intconst);
  return (cc.i1);
}

Int GetImin (CupitConst cc)
{
  _assert (cc.tag == Intervalconst);
  return (cc.i1);
}

Int GetImax (CupitConst cc)
{
  _assert (cc.tag == Intervalconst);
  return (cc.i2);
}

Real GetRval (CupitConst cc)
{
  _assert (cc.tag == Realconst);
  return (cc.r1);
}

Real GetRmin (CupitConst cc)
{
  _assert (cc.tag == Realervalconst);
  return (cc.r1);
}

Real GetRmax (CupitConst cc)
{
  _assert (cc.tag == Realervalconst);
  return (cc.r2);
}

CupitConst ComputeConst (tOilOp op, CupitConst cc1, CupitConst cc2)
{
  CupitConst r;  /* result */
  int        nr = OilOpName(op);
#define BoolTag   r.tag = Intconst
  r.tag = cc1.tag;
  switch (nr) {
    case bAndOp_name:     r.i1 = cc1.i1 && cc2.i1; break;
    case bOrOp_name:      r.i1 = cc1.i1 || cc2.i1; break;
    case bXorOp_name:     r.i1 = cc1.i1 != cc2.i1; break;
    case iEqOp_name:      r.i1 = cc1.i1 == cc2.i1; break;
    case iNeqOp_name:     r.i1 = cc1.i1 != cc2.i1; break;
    case rEqOp_name:      r.i1 = cc1.r1 == cc2.r1; BoolTag; break;
    case rNeqOp_name:     r.i1 = cc1.r1 != cc2.r1; BoolTag; break;
    case IntervalEqOp_name:
      r.i1 = (cc1.i1 == cc2.i1) && (cc1.i2 == cc2.i2); BoolTag; break;
    case IntervalNeqOp_name:
      r.i1 = (cc1.i1 != cc2.i1) || (cc1.i2 != cc2.i2); BoolTag; break;
    case RealervalEqOp_name:
      r.i1 = (cc1.r1 == cc2.r1) && (cc1.r2 == cc2.r2); BoolTag; break;
```

```
case RealervalNeqOp_name:
  r.i1 = (cc1.r1 != cc2.r1) || (cc1.r2 != cc2.r2); BoolTag; break;
case iLtOp_name:      r.i1 = cc1.i1 < cc2.i1; break;
case rLtOp_name:      r.i1 = cc1.r1 < cc2.r1; BoolTag; break;
case iGtOp_name:      r.i1 = cc1.i1 > cc2.i1; break;
case rGtOp_name:      r.i1 = cc1.r1 > cc2.r1; BoolTag; break;
case iLeOp_name:      r.i1 = cc1.i1 <= cc2.i1; break;
case rLeOp_name:      r.i1 = cc1.r1 <= cc2.r1; BoolTag; break;
case iGeOp_name:      r.i1 = cc1.i1 >= cc2.i1; break;
case rGeOp_name:      r.i1 = cc1.r1 >= cc2.r1; BoolTag; break;
case IntInOp1_name:
case IntInOp2_name:
case IntInOp_name:
  r.i1 = (cc1.i1 >= cc2.i1) && (cc1.i1 <= cc2.i2); BoolTag; break;
case RealInOp_name:
  r.i1 = (cc1.r1 >= cc2.r1) && (cc1.r1 <= cc2.r2); BoolTag; break;
case IntIntervalOp1_name:
case IntIntervalOp2_name:
case IntIntervalOp_name:
  r.i1 = cc1.i1;  r.i2 = cc2.i1; r.tag = Intervalconst; break;
case RealIntervalOp_name:
  r.r1 = cc1.r1;  r.r2 = cc2.r1; r.tag = Realervalconst; break;
case iBitandOp_name:  r.i1 = cc1.i1 & cc2.i1; break;
case iBitorOp_name:   r.i1 = cc1.i1 | cc2.i1; break;
case iBitxorOp_name:  r.i1 = cc1.i1 ^ cc2.i1; break;
case iLshiftOp_name:  r.i1 = cc1.i1 << cc2.i1; break;
case iRshiftOp_name:  r.i1 = cc1.i1 >> cc2.i1; break;
case iPlusOp_name:    r.i1 = cc1.i1 + cc2.i1; break;
case rPlusOp_name:    r.r1 = cc1.r1 + cc2.r1; break;
case iMinusOp_name:   r.i1 = cc1.i1 - cc2.i1; break;
case rMinusOp_name:   r.r1 = cc1.r1 - cc2.r1; break;
case iMulOp_name:     r.i1 = cc1.i1 * cc2.i1; break;
case rMulOp_name:     r.r1 = cc1.r1 * cc2.r1; break;
case iDivOp_name:     r.i1 = cc1.i1 / cc2.i1; break;
case rDivOp_name:     r.r1 = cc1.r1 / cc2.r1; break;
case iModOp_name:     r.i1 = cc1.i1 % cc2.i1; break;
case rModOp_name:     r.r1 = fmod (cc1.r1, cc2.r1); break;
case iExpOp_name:
  r.i1 = 1;
  while (cc2.i1 > 0) {
    if (cc2.i1 & 1 == 0)  /* square */
      r.i1 *= r.i1,    cc2.i1 >>= 1;
    else                  /* and multiply */
      r.i1 *= cc1.i1,  cc2.i1--;
  }
  break;
case riExpOp_name:
  r.r1 = 1.0;
  while (cc2.i1 > 0) {
    if (cc2.i1 & 1 == 0)  /* square */
      r.r1 *= r.r1,    cc2.i1 >>= 1;
    else                  /* and multiply */
      r.r1 *= cc1.r1,  cc2.i1--;
  }
  break;
```

```
    case rExpOp_name:
      r.r1 = pow (cc1.r1, cc2.r1); break;
    case iNegOp_name:     r.i1 = -cc1.i1; break;
    case rNegOp_name:     r.r1 = -cc1.r1; break;
    case bNotOp_name:     r.i1 = !cc1.i1; break;
    case iBitnotOp_name:  r.i1 = ~cc1.i1; break;
    case iMinOp_name:
    case iMinOp1_name:
    case iMinOp2_name:    r.i1 = cc1.i1; break;
    case rMinOp_name:     r.r1 = cc1.r1; break;
    case iMaxOp_name:
    case iMaxOp1_name:
    case iMaxOp2_name:    r.i1 = cc1.i2; break;
    case rMaxOp_name:     r.r1 = cc1.r2; break;
    default: fprintf (stderr, "Operator number: %d   ", nr);
             _assert (false);
  }
#undef BoolTag
  return (r);
}

char* Const2Str (CupitConst cc)
{
  char *res = malloc (40);
  switch (cc.tag) {
    case Intconst:        sprintf (res, "%d", cc.i1); break;
    case Realconst:       sprintf (res, "%g", cc.r1); break;
    case Intervalconst:   sprintf (res, "%d...%d", cc.i1, cc.i2); break;
    case Realervalconst:  sprintf (res, "_RealIntervalOp(%g,%g)",
                                   cc.r1, cc.r2); break;
    case Errorconst:      return ("<CupitConst Errorconst>");
    default: _assert (false); return ("<garbage CupitConst>");
  }
  return (res);
}
}
```

This macro is attached to an output file.

22.3 Ternary expressions

For ternary expressions the condition has to have `Bool` type, and the types of the two result expressions must be the same.

170 *Ternary Expression Analysis*[170] ≡

```
{
 RULE rTernaryExpr :
   Expr ::=  Expr '?' Expr ':' Expr
 COMPUTE
   Expr[1].Type =
     Oil2DefTbl (OilGetArgType (
       OilIdOp2 (TernOp, DefTbl2Oil(Expr[3].Type),
                 DefTbl2Oil (Expr[4].Type)), 0));
   Expr[1].Access = ConstAcc;
   Expr[1].MayBeRead = DSunite (DSunite (
                        DSunite (Expr[2].MayBeRead, Expr[2].MayBeUsed),
```

```
                        DSunite (Expr[3].MayBeRead, Expr[3].MayBeUsed)),
                        DSunite (Expr[4].MayBeRead, Expr[4].MayBeUsed));
  Expr[1].Kind =
    IF (AND (AND (EQ (Expr[2].Kind, ConstantK),
        EQ (Expr[3].Kind, ConstantK)), EQ (Expr[4].Kind, ConstantK)),
      /* then */ ConstantK,  /* else */ VariableK);
  Expr[1].Val =
    IF (EQ (Expr[1].Kind, VariableK),
      ErrorConst,
      IF (EQ (GetIval (Expr[2].Val), 0),
        /* then */ Expr[4].Val, /* else */ Expr[3].Val));
  Expr[1].known =
    IF (AND (NE (Expr[2].Type, BoolKey), NE (Expr[2].Type, NoKey)),
      Message2 (ERROR, "Conditional expression of '?:' has wrong type: %s %s",
                SymString (GetSym (Expr[2].Type, NoSym)),
                "(expected: Bool)"),
    /* else */
    IF (AND (AND (EQ (Expr[1].Type, NoKey),
        NE (Expr[3].Type, NoKey)), NE (Expr[4].Type, NoKey)),
      Message2 (ERROR, "Illegal result type pair %s/%s for '?:' operator",
                SymString (GetSym (Expr[3].Type, NoSym)),
                SymString (GetSym (Expr[4].Type, NoSym)))))
    DEPENDS_ON TAIL.known;
END;
}
```

This macro is invoked in definition 144.

22.4 Binary expressions

The compatibility of operands of binary expressions is checked using the operations provided by the Oil declarations given in section 22.1. For constant expressions, constant folding is performed using the `ComputeConst` operation defined in section 22.2.

Binary Expression Analysis[171] ≡ 171

```
{
RULE rBinaryExpr :
  Expr ::=  Expr BinOp Expr
COMPUTE
  Expr[1].Operator = OilIdOp2 (BinOp.Operator, DefTbl2Oil(Expr[2].Type),
                               DefTbl2Oil(Expr[3].Type));
  Expr[1].Type = Oil2DefTbl (OilGetArgType (Expr[1].Operator, 0));
  GETSUBCOORD (2);
  Expr[1].Access = ConstAcc;
  Expr[1].MayBeRead = DSunite (DSunite (Expr[2].MayBeRead, Expr[2].MayBeUsed),
                               DSunite (Expr[3].MayBeRead, Expr[3].MayBeUsed));
  Expr[1].Kind =
    IF (AND (AND (AND (EQ (Expr[2].Kind, ConstantK),
        EQ (Expr[3].Kind, ConstantK)),
        NE (GetKind (Expr[2].Type, SymbolicTypeK), SymbolicTypeK)),
        NE (GetKind (Expr[3].Type, SymbolicTypeK), SymbolicTypeK)),
      ConstantK, /* else */ VariableK) DEPENDS_ON Expr[1].known;
  Expr[1].Val =
    IF (EQ (Expr[1].Kind, ConstantK),
      ComputeConst (Expr[1].Operator, Expr[2].Val, Expr[3].Val),
      /* no constant folding defined for symbolic types */
```

```
        ErrorConst);
      Expr[1].known =
        IF (AND (AND (EQ (Expr[1].Type, NoKey),
            NE(Expr[2].Type, NoKey)), NE (Expr[3].Type, NoKey)),
          ORDER (
            Message2 (ERROR, "Illegal operand types %s/%s for binary operator",
                      SymString (GetSym (Expr[2].Type, NoSym)),
                      SymString (GetSym (Expr[3].Type, NoSym))),
            true)) DEPENDS_ON Expr[1].known;
    END;
    }
```

This macro is defined in definitions 171 and 173.
This macro is invoked in definition 144.

The phrases for the individual operators are handled with the following parameterized macro; it is also used for unary operators below.

172 *Operator Mapping*[172](◇3) ≡

```
{RULE r◇1: ◇2 ::= ◇3 COMPUTE
  ◇2.Operator = ◇1;
END;
}
```

This macro is invoked in definitions 126, 126, 126, 126, 126, 126, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 173, 175, 175, 175, 175, 175, and 175.

173 *Binary Expression Analysis*[173] ≡

```
{
 Operator Mapping[172](‘OrOp’,‘   BinOp’,‘ ’OR’’)
 Operator Mapping[172](‘XorOp’,‘  BinOp’,‘ ’XOR’’)
 Operator Mapping[172](‘AndOp’,‘  BinOp’,‘ ’AND’’)
 Operator Mapping[172](‘EqOp’,‘   BinOp’,‘ ’=’’)
 Operator Mapping[172](‘NeqOp’,‘  BinOp’,‘ ’<>’’)
 Operator Mapping[172](‘LtOp’,‘   BinOp’,‘ ’<’’)
 Operator Mapping[172](‘GtOp’,‘   BinOp’,‘ ’>’’)
 Operator Mapping[172](‘LeOp’,‘   BinOp’,‘ ’<=’’)
 Operator Mapping[172](‘GeOp’,‘   BinOp’,‘ ’>=’’)
 Operator Mapping[172](‘InOp’,‘   BinOp’,‘ ’IN’’)
 Operator Mapping[172](‘BitorOp’,‘    BinOp’,‘ ’BITOR’’)
 Operator Mapping[172](‘BitxorOp’,‘   BinOp’,‘ ’BITXOR’’)
 Operator Mapping[172](‘BitandOp’,‘   BinOp’,‘ ’BITAND’’)
 Operator Mapping[172](‘IntervalOp’,‘ BinOp’,‘ ’...’’)
 Operator Mapping[172](‘LshiftOp’,‘   BinOp’,‘ ’LSHIFT’’)
 Operator Mapping[172](‘RshiftOp’,‘   BinOp’,‘ ’RSHIFT’’)
 Operator Mapping[172](‘PlusOp’,‘     BinOp’,‘ ’+’’)
 Operator Mapping[172](‘MinusOp’,‘    BinOp’,‘ ’-’’)
 Operator Mapping[172](‘MulOp’,‘      BinOp’,‘ ’*’’)
 Operator Mapping[172](‘DivOp’,‘      BinOp’,‘ ’/’’)
 Operator Mapping[172](‘ModOp’,‘      BinOp’,‘ ’%’’)
 Operator Mapping[172](‘ExpOp’,‘      BinOp’,‘ ’**’’)
}
```

This macro is defined in definitions 171 and 173.
This macro is invoked in definition 144.

22.5 Unary expressions

Unary operators are handled analogous to the binary ones.

174 *Unary Expression Analysis*[174] ≡

```
{
RULE rUnaryExpr :
  Expr ::=  UnaryOp Expr
COMPUTE
  Expr[1].Operator = OilIdOp1 (UnaryOp.Operator, DefTbl2Oil(Expr[2].Type));
  Expr[1].Type = Oil2DefTbl (OilGetArgType (Expr[1].Operator, 0));
  Expr[1].known =
    IF (AND (EQ (Expr[1].Type, NoKey), NE(Expr[2].Type, NoKey)),
      Message1 (ERROR, "Illegal operand type %s for unary operator",
                SymString (GetSym (Expr[2].Type, NoSym))))
    DEPENDS_ON Expr[2].known;
  Expr[1].Access = ConstAcc;
  Expr[1].MayBeRead = DSunite (Expr[2].MayBeRead, Expr[2].MayBeUsed);
  Expr[1].Kind =
    IF (AND (EQ (Expr[2].Kind, ConstantK), NE (UnaryOp.Operator, RandomOp)),
      ConstantK, /* else */  VariableK);
  Expr[1].Val = IF (EQ (Expr[1].Kind, ConstantK),
                  ComputeConst (Expr[1].Operator, Expr[2].Val, ErrorConst),
                  ErrorConst);
END;
}
```

This macro is defined in definitions 174, 175, and 177.
This macro is invoked in definition 144.

Unary Expression Analysis[175] ≡ 175

```
{
Operator Mapping[172](`NotOp',`    UnaryOp',` 'NOT'')
Operator Mapping[172](`BitnotOp',` UnaryOp',` 'BITNOT'')
Operator Mapping[172](`NegOp',`    UnaryOp',` '-'')
Operator Mapping[172](`MinOp',`    UnaryOp',` 'MIN'')
Operator Mapping[172](`MaxOp',`    UnaryOp',` 'MAX'')
Operator Mapping[172](`RandomOp',` UnaryOp',` 'RANDOM'')
}
```

This macro is defined in definitions 174, 175, and 177.
This macro is invoked in definition 144.

Which type conversions are available is computed by testing the `Params` property of the type to convert to: If there is to conversion at all, we emit an error message. If there is a conversion, but with different arguments, the situation will be handled just like parameter/argument mismatches in a function call.

Type Conversion Expression Analysis[176] ≡ 176

```
{
RULE rTypeConvExpr :
  Expr ::=  TypeId '(' ExprList ')'
COMPUTE
  CHAINSTART ExprList.Paramcounter = 0;
  Expr.Type = TypeId.Key;
  Expr[1].Access = ConstAcc;
  Expr[1].Kind = VariableK;
  Expr[1].Val = ErrorConst;
  .MayBeUsed = ExprList CONSTITUENTS Expr.MayBeUsed
               WITH (DefTblKeySet, DSunite, IDENTICAL, DSempty);
  .MayBeRead = ExprList CONSTITUENTS Expr.MayBeRead
               WITH (DefTblKeySet, DSunite, IDENTICAL, DSempty);
  Expr[1].MayBeRead = DSunite (.MayBeUsed, .MayBeRead);
  ExprList.InhParams = GetParams (TypeId.Key, NoKeyArray)
```

```
      DEPENDS_ON Expr.known;
    Expr.known =
      IF (EQ (ExprList.InhParams, NoKeyArray),
        Message1 (ERROR, "No constructor available for %s",
                  SymString (TypeId.Sym)))
      DEPENDS_ON TAIL.known;
  END;
  }
```
This macro is invoked in definition 144.

The **MAXINDEX** expression is defined for networks, node groups, arrays, and connection interfaces only:

177 *Unary Expression Analysis*[177] ≡

```
  {
  RULE rMaxindexExpr :
    Expr ::=  'MAXINDEX' '(' Object ')'
  COMPUTE
    .Kind = GetKind (Object.Type, UndefinedK) DEPENDS_ON Expr.known;
    Expr.Type = IntKey;
    Expr.Access = ConstAcc;
    Expr.Kind = VariableK;
    Expr.Val = ErrorConst;
    Expr.MayBeRead = DSunite (Object.MayBeUsed, Object.MayBeRead);
    Expr.known =
      IF (AND (AND (AND (AND (NE (GetKind (Object.Type, NetTypeK), NetTypeK),
          NE (GetKind (Object.Type, ArrayTypeK), ArrayTypeK)),
          NE (GetKind (Object.Type, NodeArrayTypeK), NodeArrayTypeK)),
          NE (GetKind (Object.Type, NodeGroupTypeK), NodeGroupTypeK)),
          OR (NE (GetKind (Object.Type, ConTypeK), ConTypeK),
              NE (Object.Kind, ParVariableK))),
        Message (ERROR, "MAXINDEX is not defined for this kind of object"),
      /* else */
      IF (AND (EQ (GetKind (Object.Type, UndefinedK), NetTypeK),
          NE (Context Kind[111], GlobalSubroutineContext)),
        Message (ERROR, "MAXINDEX(network) allowed in central agent only")))
      DEPENDS_ON Object.known;
  END;
  }
```
This macro is defined in definitions 174, 175, and 177.
This macro is invoked in definition 144.

22.6 Denoters

The type of denoters is always determined from their appearance. The access of a denoter is always `CONST`.

178 *Denoter Expression Analysis*[178] ≡

```
  {
  RULE rDenoterExpr :
    Expr ::=  Denoter
  COMPUTE
    TRANSFER Type, Val;
    Expr.Access = ConstAcc;
    Expr.Kind = ConstantK;
    Expr.MayBeRead = NoDefTblKeySet;
  END;
```

```
}
```

This macro is defined in definitions 178, 179, 180, and 181.
This macro is invoked in definition 144.

For integer denoters, the exact type is determined from the value denoted: The assigned type is always the smallest one possible (except for the maximal negative value since negation is an operation, e.g. -128 will have type `Int2`). The value of an integer denoter is stored in the `Sym` attribute.

Denoter Expression Analysis[179] ≡ 179

```
{
 RULE rIntDenoter :
   Denoter ::=  IntegerDenoter
 COMPUTE
   Denoter.Val = SetIval (IntegerDenoter.Sym);
   Denoter.Type =
     IF (LE (IntegerDenoter.Sym, 127),    Int1Key,
     IF (LE (IntegerDenoter.Sym, 32767),  Int2Key,
                                          IntKey)) DEPENDS_ON Denoter.known;
 END;
 }
```

This macro is defined in definitions 178, 179, 180, and 181.
This macro is invoked in definition 144.

For `Real` denoters the value has to be computed from the string representation. The `Sym` attribute contains a handle to that string.

Denoter Expression Analysis[180] ≡ 180

```
{
 RULE rRealDenoter :
   Denoter ::=  RealDenoter
 COMPUTE
   Denoter.Type = RealKey;
   Denoter.Val = SetRval (atof (SymString (RealDenoter.Sym)));
 END;
 }
```

This macro is defined in definitions 178, 179, 180, and 181.
This macro is invoked in definition 144.

For string denoters nothing special needs to be done at all, the actual value will be retrieved from the `Sym` attribute at code generation time.

Denoter Expression Analysis[181] ≡ 181

```
{
 RULE rStringDenoter :
   Denoter ::=  StringDenoter
 COMPUTE
   Denoter.Type = StringKey;
   Denoter.Val  = ErrorConst;  /* no computation on strings */
 END;
 }
```

This macro is defined in definitions 178, 179, 180, and 181.
This macro is invoked in definition 144.

22.7 Function calls

Function calls are similar to procedure calls except that (1) we have to process the return type and (2) function calls are restricted to remain on the same level of parallelism, e.g. we may not call a connection function from a node subroutine.

Function Call Analysis[182] ≡ 182

```
{
 RULE rFCallExpr :
   Expr ::=  FunctionCall
 COMPUTE
   Expr.Type = FunctionCall.Type;
   Expr.Access = ConstAcc;
   Expr.Kind = VariableK;
   Expr.Val = ErrorConst;
   Expr.MayBeRead = FunctionCall.MayBeRead;
 END;

 RULE rObjFCallExpr:
   Expr ::=  ObjectFunctionCall
 COMPUTE
   Expr.Type = ObjectFunctionCall.Type;
   Expr.Access = ConstAcc;
   Expr.Kind = VariableK;
   Expr.Val = ErrorConst;
   Expr.MayBeRead = ObjectFunctionCall.MayBeRead;
 END;

 RULE rFunctionCall :
   FunctionCall ::=  FunctionId '(' ArgumentList ')'
 COMPUTE
   CHAINSTART ArgumentList.Paramcounter = 0;
   ArgumentList.InhParams = GetParams (FunctionId.Key, NoKeyArray)
     DEPENDS_ON FunctionCall.known;
   FunctionCall.Type = GetType (FunctionId.Key, NoKey)
     DEPENDS_ON FunctionCall.known;
   .nr = KeyArraySize (GetParams (FunctionId.Key, NoKeyArray))
         DEPENDS_ON FunctionCall.known;
   FunctionCall.known =
     IF (AND (EQ (FunctionId.Kind, FunctionK),
         NE (.nr, ArgumentList.Paramcounter)),
         Message2 (ERROR, "Function call has %d parameters, expected: %d",
                   ArgumentList.Paramcounter, .nr))
     DEPENDS_ON (.done1, .done2);
   FunctionCall.CentralAgent =
     GetCentralAgent (FunctionId.Key, false) DEPENDS_ON FunctionCall.known;
   FunctionCall.MayBeRead = ArgumentList.MayBeRead; /* write is forbidden! */
   .done1 =
     IF (AND (FunctionCall.CentralAgent,
         NE (Context Kind[111], GlobalSubroutineContext)),
       Message1 (ERROR, "%s belongs to central agent, cannot be called here",
                 SymString (FunctionId.Sym)));
   .IsUsed = IF (FunctionCall.CentralAgent, used, BITOR (used, usedA));
   .done2 =
     SetIsUsed (FunctionId.Key, .IsUsed,
                BITOR (GetIsUsed (FunctionId.Key, used0), .IsUsed));
 END;

 RULE rFunctionId :
   FunctionId ::=  LowercaseIdent
 COMPUTE
```

```
  FunctionId.Kind = GetKind (FunctionId.Key, UndefinedK)
  DEPENDS_ON FunctionId.known;
  FunctionId.known =
    IF (EQ (FunctionId.Kind, ProcedureK),
      Message (ERROR, "Procedure called in value context (i.e. as a function)"),
    /* else */
    IF (NE (GetKind (FunctionId.Key, FunctionK), FunctionK),
      Message1 (ERROR, "%s is called as a global function but it is none",
                SymString (FunctionId.Sym))))
    DEPENDS_ON FunctionId.known;
END;

RULE rObjectFunctionCall1 :
  ObjectFunctionCall ::=  Object '.' ObjectFunctionId '(' ArgumentList ')'
COMPUTE
  CHAINSTART ArgumentList.Paramcounter = 0;
  ObjectFunctionId.ScopeKey = Object.Type;
  ArgumentList.InhParams = GetParams (ObjectFunctionId.Key, NoKeyArray)
    DEPENDS_ON ObjectFunctionCall.known;
  ObjectFunctionCall.Type = GetType (ObjectFunctionId.Key, NoKey)
    DEPENDS_ON ObjectFunctionCall.known;
  ObjectFunctionCall.MayBeRead =
    DSunite (ArgumentList.MayBeRead,
             GetMayBeRead (ObjectFunctionId.Key, NoDefTblKeySet))
    DEPENDS_ON ObjectFunctionCall.known;
  .nr = KeyArraySize (GetParams (ObjectFunctionId.Key, NoKeyArray))
        DEPENDS_ON ObjectFunctionCall.known;
  ObjectFunctionCall.known =
    IF (AND (EQ (ObjectFunctionId.Kind, FunctionK),
        NE (.nr, ArgumentList.Paramcounter)),
      Message2 (ERROR, "Object function call has %d parameters, expected: %d",
                ArgumentList.Paramcounter, .nr))
    DEPENDS_ON (.done1, .done2, .done3);
  .done1 =
    SetIsUsed (ObjectFunctionId.Key, used,
               BITOR (GetIsUsed (ObjectFunctionId.Key, used0), used));
  .done2 =
    IF (OR (EQ (Object.Kind, ParVariableK), EQ (Object.Kind, ParVariableSelK)),
      Message (ERROR, "Function calls are impossible for parallel variables"));
  .done3 =
    IF (AND (NE (Subroutine ME Type[109], Object.Type),
        NE (Object.Type, NoKey)),
      Message2 (ERROR, "%s function calls are allowed in %s subroutines only",
                SymString (GetSym (Object.Type, NoSym)),
                SymString (GetSym (Object.Type, NoSym))));
END;

SYMBOL ObjectFunctionId INHERITS FieldUse, NoKeyMsg, NameOccurrence END;

RULE rObjectFunctionId :
  ObjectFunctionId ::=  LowercaseIdent
COMPUTE
  ObjectFunctionId.Kind = GetKind (ObjectFunctionId.Key, UndefinedK)
  DEPENDS_ON ObjectFunctionId.known;
  ObjectFunctionId.known =
```

```
    IF (EQ (ObjectFunctionId.Kind, ProcedureK),
      Message (ERROR, "Procedure called in value context (i.e. as a function)"),
    /* else */
    IF (NE (ObjectFunctionId.Kind, FunctionK),
      Message2 (ERROR, "%s is not a FUNCTION in type %s",
                SymString (ObjectFunctionId.Sym),
                SymString (GetSym (ObjectFunctionId.ScopeKey, NoSym)))))
    DEPENDS_ON ObjectFunctionId.known;
END;
}
```

This macro is invoked in definition 144.

23 Objects

The next few sections discuss the various ways to refer to objects. We first describe the use of an object in an expression and set some defaults for object processing.

183 *Object Expression Analysis*[183] ≡

```
{
RULE rObjectExpr :
  Expr ::=  Object
COMPUTE
  TRANSFER Type, Kind, Access, Val;
  Expr.MayBeRead = Object.MayBeRead;
  Expr.MayBeUsed = Object.MayBeUsed;
  Expr.known =
    IF (OR (EQ (Object.Kind, ParVariableK), EQ (Object.Kind, ParVariableSelK)),
      Message (ERROR, "Parallel variable cannot be used in expression"))
    DEPENDS_ON Object.known;
END;

SYMBOL Object COMPUTE  /* default values: */
  THIS.Val  = ErrorConst;
  THIS.Kind = VariableK;
  THIS.Mode = NoMode;
  THIS.ParVarType = NoKey;
  THIS.ParVarMode = NoMode;
  THIS.MayBeUsed  = NoDefTblKeySet;
  THIS.MayBeRead  = NoDefTblKeySet;
END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

23.1 ME, YOU, INDEX, and explicit variables

The name of a network variable may be used only in global subroutines (which then automatically become part of the central agent).

184 *Object Expression Analysis*[184] ≡

```
{
RULE rDirectObject :
  Object ::=  Objectname
COMPUTE
```

```
    TRANSFER Access;
    Object.Val  = Objectname.Val;
    Object.Kind = Objectname.Kind;
    Object.Type =
      IF (AND (EQ (GetKind (Objectname.Type, UndefinedK), NetTypeK),
          NE (Context Kind[111], GlobalSubroutineContext)),
        ORDER (
          Message (ERROR, "Network variables cannot be used explicitly here"),
          Messag2 (NOTE,  "Context Kind: %d,  should be %d",
                    Context Kind[111], GlobalSubroutineContext),
          NoKey),
        Objectname.Type)
      DEPENDS_ON Object.known;
  END;
  }
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

The object name itself may be the name of a compile-time constant, in which case we create a `ConstantK` object providing the constant value. If the name is not a constant or variable name, we emit an error message.

Object Expression Analysis[185] ≡ 185

```
 {
  RULE rObjectname :
    Objectname ::=  LowercaseIdent
  COMPUTE
    .Kind = GetKind (Objectname.Key, UndefinedK) DEPENDS_ON Objectname.known;
    Objectname.Kind =
      IF (EQ (.Kind, ConstantK),
          ConstantK,
      /* else */
      IF (EQ (.Kind, VariableK),
          VariableK,
      /* else */
      IF (NE (Objectname.Key, NoKey),
         ORDER (
           Message (ERROR, "Subroutine name used as data object name"),
           UndefinedK),
         UndefinedK)));
    Objectname.Type   = GetType (Objectname.Key, NoKey)
      DEPENDS_ON Objectname.known;
    Objectname.Access = GetAccess (Objectname.Key, VarAcc)
      DEPENDS_ON Objectname.known;
    Objectname.Val    = GetVal (Objectname.Key, ErrorConst)
      DEPENDS_ON Objectname.known;
    Objectname.CentralAgent =
      EQ (GetKind (Objectname.Type, UndefinedK), NetTypeK);
    Objectname.known = Objectname.Kind;
  END;
  }
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

The object `ME` has `CONST` access rights in reduction and winner-takes-all functions and `VAR` rights otherwise. It is not available in global contexts. `YOU` is always `CONST` and is available only in the body of `MERGE` procedures and `REDUCTION` or `WTA` functions.

186 *Object Expression Analysis*[186] ≡

```
{
 RULE rMeObject :
   Object ::=  'ME'
 COMPUTE
   Object.Type = Subroutine ME Type[109] DEPENDS_ON Object.known;
   Object.Access =
     IF (OR (EQ (Context Kind[111], ObjSubroutineContext),
         EQ (Context Kind[111], MergeContext)),
       /* then */ VarPAcc,  /* else */ ConstPAcc);
   Object.Kind =  IF (EQ (Object.Type, VoidKey), UndefinedK, VariableK);
   Object.known =
     IF (EQ (Object.Type, VoidKey),
       Message (ERROR, "'ME' is not available in global procedures"))
     DEPENDS_ON Object.known;
 END;

 RULE rYouObject :
   Object ::=  'YOU'
 COMPUTE
   Object.Type = Subroutine YOU Type[110];
   Object.Access = ConstPAcc;
   Object.Kind =  IF (EQ (Object.Type, VoidKey), UndefinedK, VariableK);
   Object.known =
     IF (EQ (Object.Type, VoidKey),
       Message (ERROR, "'YOU' is available in MERGE/REDUCTION/WTA only"))
     DEPENDS_ON Object.known;
 END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

The object `INDEX` is available in node and network subroutines only, it always has `Int` type and `CONST` access:

187 *Object Expression Analysis*[187] ≡

```
{
 RULE rIndexObject :
   Object ::=  'INDEX'
 COMPUTE
   .Type = IF (EQ (Context Kind[111], ObjSubroutineContext),
               Subroutine ME Type[109], VoidKey) DEPENDS_ON Object.known;
   .Kind = GetKind (.Type, UndefinedK);
   Object.Type   = IntKey;
   Object.Access = ConstAcc;
   Object.known =
     IF (AND (NE (.Kind, NodeTypeK), NE (.Kind, NetTypeK)),
       Message(ERROR, "INDEX is available in network and node subroutines only"))
     DEPENDS_ON Object.known;
 END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

23.2 Selection

(1) Selection from an ordinary variable (`VariableK`) creates an ordinary variable. (2) Selection from a parallel variable (`ParVariableK`) creates a parallel variable selection (`ParVariableSelK`). (3) Selection from a parallel variable selection is not allowed. (4) Selection returns an object of the type of the element selected.

Object Expression Analysis[188] ≡ 188

```
{
 RULE rSelectionObject :
   Object ::=  Object '.' Elementname
 COMPUTE
   .Kind  = GetKind (Object[2].Type, UndefinedK) DEPENDS_ON Object[1].known;
   .Kind2 = Object[2].Kind DEPENDS_ON Object[1].known;
   Object[1].Kind =
     /* (1): */
     IF (EQ (.Kind2, VariableK),
       VariableK,
     /* else (2): */
     IF (EQ (.Kind2, ParVariableK),
       ParVariableSelK,
     /* else (3): */
     IF (EQ (.Kind2, ParVariableSelK),
       ORDER (
         Message (ERROR, "You cannot select from a parallel variable selection"),
         UndefinedK),
       UndefinedK))) DEPENDS_ON Object[1].known;
   /* (4): */
   Elementname.ScopeKey = Object[2].Type;
   Object[1].Type  = Elementname.Type;
   Object[1].Access = Object[2].Access;
   Object[1].Mode = GetMode (Elementname.Key, NoMode) DEPENDS_ON Object[1].known;
   Object[1].ParVarType = Object[2].ParVarType;
   Object[1].ParVarMode = Object[2].ParVarMode;
   Object[1].MayBeUsed =
     IF (EQ (Object[2].Type, Subroutine ME Type[109]),
       DSinsert (Object[2].MayBeUsed, Elementname.Key),
       Object[2].MayBeUsed);
   Object[1].MayBeRead = Object[2].MayBeRead;
   Messag3 (NOTE, "%s %s from type %s",
            SymString (GetSym (Object[1].Type, NoSym)),
            SymString (Elementname.Sym),
            SymString (GetSym (Elementname.ScopeKey, NoSym)));
   Object[1].known = Object[1].Kind DEPENDS_ON Elementname.known;
 END;

 SYMBOL Elementname INHERITS FieldUse, NoKeyMsg END;

 RULE rElementname :
   Elementname ::= LowercaseIdent
 COMPUTE
   TRANSFER Sym;
   Elementname.Type = GetType (Elementname.Key, NoKey)
     DEPENDS_ON Elementname.known;
   Elementname.Mode = GetMode (Elementname.Key, NoMode)
     DEPENDS_ON Elementname.known;
```

```
  Elementname.known =
    IF (AND (NE (Elementname.ScopeKey, NoKey), EQ (Elementname.Key, NoKey)),
      Message2 (ERROR, "No element %s found in type %s",
                SymString (Elementname.Sym),
                SymString (GetSym (Elementname.ScopeKey, NoSym))))
    DEPENDS_ON Elementname.known;
END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

23.3 Subscription

(1) For index expressions, the indexed object must be an array or group, an explicit network variable, or (for the all-slice only) a connection interface. (2) The resulting type is the base type of the array or group or the type of the network or the connection interface. (3) The subscripts must be integer or integer interval. (4) Subscription with slices produces parallel variables of network, node, or connection type and (5) is not allowed for other types. (6) Parallel variables cannot be subscribed, (7) I/O objects cannot be subscribed.

189 *Object Expression Analysis*[189] ≡

```
{
RULE rIndexedObject :
  Object ::=  Object '[' Expr ']'
COMPUTE
  .Kind =  /* of object type! */
    GetKind (Object[2].Type, ArrayTypeK) DEPENDS_ON Object[1].known;
  /* (7): */
  IF (IsIoAcc (Object[2].Access),
    Message (ERROR, "IO objects cannot be indexed"),
  /* else (1): */
  IF (NOT (OR (OR (OR (EQ (.Kind, ArrayTypeK), EQ (.Kind, NodeArrayTypeK)),
        EQ (.Kind, NodeGroupTypeK)), EQ (.Kind, NetTypeK))),
    Message1 (ERROR, "%s is no ARRAY/GROUP/NETWORK type, cannot be indexed",
              SymString (GetSym (Object[2].Type, NoSym))),
  /* else (6): */
  IF (OR (EQ (Object[2].Kind, ParVariableK),
      EQ (Object[2].Kind, ParVariableSelK)),
    Message (ERROR, "Object is a parallel variable and cannot be indexed"))));
  /* (2): */
  Object[1].Type =
    IF (OR (OR (EQ (.Kind, ArrayTypeK), EQ (.Kind, NodeArrayTypeK)),
        EQ (.Kind, NodeGroupTypeK)),
      GetType (Object[2].Type, NoKey), /* else */ Object[2].Type)
    DEPENDS_ON Object[1].known;
  Object[1].Kind =
    IF (IsInt (Expr.Type),
      VariableK,
    /* else (4), (5): */
    IF (IsInterval (Expr.Type),
      IF (OR (OR (EQ (.Kind, NodeArrayTypeK),
          EQ (.Kind, NodeGroupTypeK)), EQ (.Kind, NetTypeK)),
        ParVariableK,
        /* else */
        ORDER (
```

```
          IF (EQ (.Kind, ArrayTypeK),
            Message1 (ERROR, "Slice indexing not allowed for type %s",
                      SymString (GetSym (Object[2].Type, NoSym)))),
          VariableK)),
    /* else (3): */
    IF (EQ (Expr.Type, NoKey),
      VariableK,
    /* else */
      ORDER (
        Message1 (ERROR, "Index expression has illegal type: %s",
                  SymString (GetSym (Expr.Type, NoSym))),
        VariableK))))
  DEPENDS_ON Object[1].known;
  Object[1].Access = Object[2].Access;
  Object[1].Mode   = Object[2].Mode;
  Object[1].ParVarType = IF (EQ (Object[1].Kind, ParVariableK),
                         Object[1].Type, /* else */ Object[2].ParVarType);
  Object[1].ParVarMode = IF (EQ (Object[1].Kind, ParVariableK),
                         Object[2].Mode, /* else */ Object[2].ParVarMode);
  Object[1].MayBeUsed = Object[2].MayBeUsed;
  Object[1].MayBeRead = DSunite (Object[2].MayBeRead,
                              DSunite (Expr.MayBeUsed, Expr.MayBeRead));
END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

For the all-slice object the processing is mostly similar although simpler since we do not have to check the expression type. Connection interfaces are allowed, non-node arrays are not.

Object Expression Analysis[190] ≡ 190

```
{
RULE rUnindexedObject :
  Object ::=  Object '[' ']'
COMPUTE
  TRANSFER Access;
  .Kind =  /* of object type! */
    GetKind (Object[2].Type, ArrayTypeK) DEPENDS_ON Object[1].known;
  Object[1].MayBeUsed = Object[2].MayBeUsed;
  Object[1].known =
    /* (1): */
    IF (NOT (OR (OR (OR (EQ (.Kind, ConTypeK), EQ (.Kind, NodeArrayTypeK)),
          EQ (.Kind, NodeGroupTypeK)), EQ (.Kind, NetTypeK))),
      Message1 (ERROR, "a %s cannot be turned into a parallel variable",
                SymString (GetSym (Object[2].Type, NoSym))),
    /* else (6): */
    IF (EQ (Object[2].Kind, ParVariableK),
      Message1 (ERROR, "A parallel variable cannot be indexed (Type: %s)",
                SymString (GetSym (Object[2].Type, NoSym))),
    IF (EQ (Object[2].Kind, ParVariableSelK),
      Message2 (ERROR, "A parallel variable selection cannot be indexed (%s%s)",
                "Type: ", SymString (GetSym (Object[2].Type, NoSym))),
    /* else (7): */
    IF (IsIoAcc (Object[2].Access),
      Message (ERROR, "IO objects cannot be indexed")))))
    DEPENDS_ON Object[2].known;
  /* (2): */
```

```
  Object[1].Type =
    IF (OR (EQ (.Kind, NodeArrayTypeK), EQ (.Kind, NodeGroupTypeK)),
      GetType (Object[2].Type, NoKey), /* else */ Object[2].Type)
    DEPENDS_ON Object[1].known;
  Object[1].Kind = ParVariableK;
  Object[1].Mode   = Object[2].Mode;
  Object[1].ParVarType = Object[1].Type;
  Object[1].ParVarMode = Object[2].Mode;
END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

23.4 Connection addressing

A direct-connection-access object is allowed on the left hand side of an assignment statement only, because it has the side effect to create the connection. This restriction is not checked! The two objects mentioned in the direct-connection-access object must be a `OUT` connection interface and a `IN` connection interface of two single nodes. These interfaces must have the same type.

191 *Object Expression Analysis*[191] ≡

```
{
RULE rWeightObject :
  Object ::=  '{' Object '-->' Object '}'
COMPUTE
  Object[1].Type =
    IF (OR (NE (GetKind (Object[2].Type, ConTypeK), ConTypeK),
        NE (Object[2].Mode, OutMode)),
      ORDER (
        Message (ERROR, "First object must be an OUT connection interface"),
        NoKey),
    /* else */
    IF (OR (NE (GetKind (Object[3].Type, ConTypeK), ConTypeK),
        NE (Object[2].Mode, InMode)),
      ORDER (
        Message (ERROR, "Second object must be an IN connection interface"),
        NoKey),
    /* else */
    IF (NE (Object[2].Type, Object[3].Type),
      ORDER (Message2 (ERROR, "Connection type conflict: %s/%s",
                      SymString (GetSym (Object[2].Type, NoSym)),
                      SymString (GetSym (Object[3].Type, NoSym))),
             NoKey),
    /* else */
      Object[2].Type)))
    DEPENDS_ON Object[1].known;
  Object[1].Access = VarAcc;
END;
}
```

This macro is defined in definitions 183, 184, 185, 186, 187, 188, 189, 190, and 191.
This macro is invoked in definition 144.

24 Put it all together

All the above specifications belong into certain files, to which they are now assigned:

The `.pdl` (property definition language) file contains the property definitions plus include file names to make the types of the properties known.

type.pdl[192] ≡ 192

{
```
"keyarray.h"  /* KeyArray */
"type.h"      /* tKind */
"oiladt2.h"   /* tOilType */
"folding.h"   /* CupitConst */
"deftblkeyset.h"  /* DefTblKeyset */
```
Type Analysis Properties[81]

}

This macro is attached to an output file.

The `.lido` file contains the attribute grammar specification for the type analysis.

type.lido[193] ≡ 193

{

Traversal Order Invariant[90]
Type Analysis Attributes[82]
Type Definition Analysis[91]
Data Object Definition Analysis[104]
Subroutine Definition Analysis[106]
Statement Analysis[124]
Expression Analysis[144]

}

This macro is attached to an output file.

The `.c` file implements the function that sets the properties of the predefined objects.

type.c[194] ≡ 194

{
```
#include "cupit.h"
#include "scope.h"    /* Declarations of predefined objects' xKey variables */
#include "pdl_gen.h"  /* Property manipulation functions and property types */
#include "oiladt2.h"
#include "OilDecls.h"
```

Set Properties of Predefined Objects[88]

}

This macro is attached to an output file.

The corresponding `.h` file declares this function and also declares the enumeration types that are introduced for type analysis.

type.h[195] ≡ 195

{
```
#ifndef type_H
#define type_H

#include "folding.h"
```

Kinds Of Objects[80]
Access Rights of Objects[83]
Interface Modes of Connection Elements[85]
Kinds Of Contexts[87]
```
extern void SetPredefObjProperties ();
```

```
#endif
}
```

This macro is attached to an output file.

The `.head` file is included when the complete generated attribute grammar evaluator is compiled. Therefore, it must include the interfaces of all operations used in the attribution.

196 **type.head**[196] ≡

```
{
#include "cupit.h"
#include "pdl_gen.h"  /* incl. keyarray.h, type.h, envmod.h, folding.h */
#include "oil_interface.h"
#include "OilDecls.h"
#include "deftblkeyset.h"
Parameter Access Handler[84]
IsUsed Values[86]
}
```

This macro is attached to an output file.

25 Usage analysis

This section contains two subsections that are concerned with the optimization of the access to remote connection data. This optimization is computed in two phases: First, we determine which connection procedures may read or write which elements of a connection object. Second, we decide which of these elements to fetch or send individually and which to package into a single larger communication operation.

The first phase is carried out during type analysis which you have already seen in the previous sections. The first subsection defines the abstract data type used by this analysis to store the results, `may-be-used sets`.

The second phase is carried out during code generation, when the actual fetch and send code for those variants of the connection procedures are generated that are used for the remote connection case. The second subsection defines the procedures that make the decision on communication aggregation and generate the corresponding code.

25.1 May-be-used sets

For the generation of code to access the data pointed to by a remote connection object, we need information about which parts of the connection object pointed to are used. Such information is collected during type analysis and stored in an object of type "set of definition table keys" (`DefTableKeySet`). This abstract data type is implemented here (in a rather simple fashion).

The operations needed are creation (`NoDefTblKeySet`, `DSempty`, `DSmk`), element insertion (`DSinsert`), set untion (`DSunite`) and set iteration (`DSiterate`, `DSnext`). We also implement an element test operation (`DScontains`).

197 **deftblkeyset.h**[197] ≡

```
{/* RCS: $Id: usage.fw,v 1.8 1994/11/07 10:49:38 prechelt Exp prechelt $ */
#ifndef DefTblKeySet_H
#define DefTblKeySet_H

#include "cupit.h"
#include "deftbl.h"

typedef struct _dtkset {
```

```
  struct _dtkset *next;
  DefTableKey    key;
} *DefTblKeySet;

#define NoDefTblKeySet ((DefTblKeySet)0)
#define DSempty()      ((DefTblKeySet)0)

DefTblKeySet DSmk (DefTableKey k);
DefTblKeySet DSinsert (DefTblKeySet s, DefTableKey k);
DefTblKeySet DSunite (DefTblKeySet s1, DefTblKeySet s2);
Bool         DScontains (DefTblKeySet s, DefTableKey k);
void         DSiterate (DefTblKeySet s);
DefTableKey  DSnext ();  /* returns NoKey at end */
void         DSprint (FILE *fp, DefTblKeySet s);

#endif
}
```

This macro is attached to an output file.

The implementation is trivially done via a linked list, because the typical size of these sets will be below 10. The sets are never destroyed.

deftblkeyset.c[198] ≡ 198

```
{/* RCS: $Id: usage.fw,v 1.8 1994/11/07 10:49:38 prechelt Exp prechelt $ */

#include <stdio.h>  /* for stderr to be used by _assert() */
#include "cupit.h"
#include "deftblkeyset.h"
#include "pdl_gen.h"

static DefTblKeySet iter;

DefTblKeySet DSmk (DefTableKey k)
{
  DefTblKeySet new;
  new = (DefTblKeySet)malloc (sizeof (struct _dtkset));
  _assert (new != NoDefTblKeySet);
  new->key = k;
  new->next = NoDefTblKeySet;
  return (new);
}

DefTblKeySet DSinsert (DefTblKeySet s, DefTableKey k)
{
  DefTblKeySet new, run = s;
  while (run != NoDefTblKeySet) {
    if (run->key == k)
      return (s);  /* k already in s; no insertion needed */
    run = run->next;
  }
  /* k not already in s; insert it at the beginning: */
  new = (DefTblKeySet)malloc (sizeof (struct _dtkset));
  _assert (new != NoDefTblKeySet);
  new->key = k;
  new->next = s;
  return (new);
```

```
}

DefTblKeySet DSunite (DefTblKeySet s1, DefTblKeySet s2)
{
  DefTblKeySet new = s1;
  while (s2 != NoDefTblKeySet) {
    new = DSinsert (new, s2->key);
    s2 = s2->next;
  }
  return (new);
}

Bool DScontains (DefTblKeySet s, DefTableKey k)
{
  while (s != NoDefTblKeySet) {
    if (s->key == k)
      return (true);
    s = s->next;
  }
  return (false);
}

void DSiterate (DefTblKeySet s)
{
  iter = s;
}

DefTableKey DSnext ()
{
  DefTblKeySet olditer = iter;
  if (iter != NoDefTblKeySet)
    iter = iter->next;
  return (olditer == NoDefTblKeySet ? NoKey : olditer->key);
}

void DSprint (FILE *fp, DefTblKeySet s)
{
  DefTblKeySet new;
  fprintf (fp, "{");
  while (s != NoDefTblKeySet) {
    new = s->next;
    fprintf (fp, "%s%s", SymString (GetSym (s->key, NoSym)),
             new != NoDefTblKeySet ? ", " : "");
    s = new;
  }
  fprintf (fp, "} ");
}
}
```

This macro is attached to an output file.

25.2 Accessing remote connection objects

By construction of our data distribution scheme, object procedures always work on local data — with one exception: Connection operations can be called for local data only from either the input or the output end of the connection. For the other end (called the *remote end*), data to be read by a procedure has to be fetched before the procedure can be called and data that was changed during the procedure call must be sent back to the original data portion of the connection object (the so called *data end*). Fetching and sending data for the whole procedure call at once simplifies code generation vastly, because it allows to use the very same procedures as for the local case, just operating on a pseudo-local object constructed from the fetched data before the call and updated back into the original object by sending data after the call.

Which data to fetch and send is determined statically by the may-be-read set of data elements that might be read by the procedure and the may-be-written set of data elements that may need to be updated. These sets can be computed by static analysis of the program text using dataflow techniques. In this compiler we use an extremely simple criterion of "textual presence" for this purpose.

In order to maintain the correct semantics of the call, two conditions must be satisfied: (1) the may-be-read and may-be-written sets must be conservative, i.e., must contain all data that is actually read or written, respectively, during the procedure call and (2) the may-be-written set must not contain any element that is not also part of the may-be-read set, because otherwise we might write back an uninitialized value (if the element was not fetched and was not actually written). This means that we always have to add all elements of the may-be-written set to the may-be-read set even those for which this is not indicated by the dataflow analysis.

The procedures to generate the code that fetches the may-be-read set and sends the may-be-written set are implemented in the following module which defines the procedures `remoteFetchCode` and `remoteSendCode`:

remoteconcomm.h[199] ≡ 199

```
{
 #ifndef remoteconcomm_H
 #define remoteconcomm_H
 #include "cupit.h"
 #include "deftblkeyset.h"
 #include "ptg_gen.h"

 void makeRemoteFetchCode (DefTblKeySet read, Bool nonoptimal, DefTableKey type,
                           Bool hidelatency, Bool highlatency);
 void makeRemoteSendCode (DefTblKeySet write, Bool nonoptimal, DefTableKey type,
                          Bool hidelatency, Bool highlatency);
 PTGNode getRemoteFetchCode ();
 PTGNode getRemoteSendCode ();
 PTGNode getRemoteCommCost ();
 #endif
}
```

This macro is attached to an output file.

The code generated by this module allows to produce multiple versions of code: optimized code, code that communicates each data element individually, code that always communicates the complete connection object, code that simulates higher latency, and code that simulates (from the view of the `timerValue` function) latency hiding. The latter version assumes that the code is inserted at the beginning of a block, because it declares a variable.

To generate efficient code, this module makes decisions about when to fetch or send individual elements and when to clump together the fetch or send operations for several elements into one communication procedure call. Such aggregation is often useful because it avoids the constant cost of a communication call (the communication latency). Obviously we should always aggregate the communication of those elements that are neighbours in the underlying data structure. But even if there is a gap containing data that need not be transfered between two data elements that must be transfered, it might be more efficient to transfer that gap instead of setting up a separate communication operation of each of the elements.

The above procedures can generate three different versions of the code: (1) with `nonoptimal` false, code with optimally aggregated communication operations is generated. (2) With `nonoptimal` true and `type`

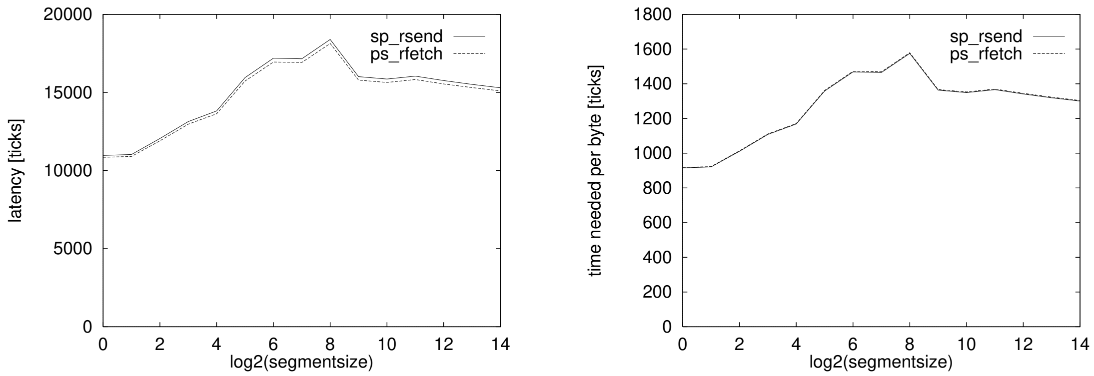
= `NoKey`, the procedures generate code to fetch or send each used element of the data type individually. (3) Otherwise, `type` must be the key of the connection type for which the procedures are called and the procedures generate code to fetch or send the whole connection object.

To decide which communication packets to aggregate, the module needs a model of the machine's communication cost. This model is constructed based on the results obtained by running the following measurement program:

200 *MasPar Communication Measurement*[200] ≡
```
{
 #define samples 10
 plural char buf[1024];
 plural char* plural pbuf = buf;

 visible program ()
 {
   int i, lxN, lyN;
   int tsend1,  tsend1024, tfetch1, tfetch1024;
   dpuTimerStart();
   sp_rsend (iproc, buf, pbuf, 1);  /* strange behavior of first dpuTimer call */
   dpuTimerTicks2();
   for (lxN = 0; lxN <= lxprocN; lxN++) {
     for (lyN = lxN; lyN <= lyprocN && lyN <= lxN+1; lyN++) {
       plural int your_xPE, your_yPE, your_PE;
       plural int x0 = (plural _bint)ixproc & (plural _bint)~_M(lxN),
                  y0 = (plural _bint)iyproc & (plural _bint)~_M(lyN);
       int segsize = _S(lxN+lyN);
       int tsend1 = 0,  tsend128 = 0,  tsend1024 = 0,
           tfetch1 = 0, tfetch128 = 0, tfetch1024 = 0;
       for (i = 0; i < samples; i++) {
         your_xPE = x0 + p_iRandomOp (p_IntIntervalOp (0, _S(lxN)));
         your_yPE = y0 + p_iRandomOp (p_IntIntervalOp (0, _S(lyN)));
         your_PE = your_xPE + (your_yPE << lxprocN);

         dpuTimerStart (); sp_rsend (your_PE, buf, pbuf, 1);
         tsend1 += dpuTimerTicks2 ();
         dpuTimerStart (); ps_rfetch (your_PE, pbuf, buf, 1);
         tfetch1 += dpuTimerTicks2 ();

         dpuTimerStart (); sp_rsend (your_PE, buf, pbuf, 1024);
         tsend1024 += dpuTimerTicks2 ();
         dpuTimerStart (); ps_rfetch (your_PE, pbuf, buf, 1024);
         tfetch1024 += dpuTimerTicks2 ();
       }
       printf ("%5d %2d   %5d %5d   %5d %5d\n",
           segsize, _log2(segsize), tsend1/samples, tfetch1/samples,
           (tsend1024-tsend1)/1023/samples, (tfetch1024-tfetch1)/1023/samples);
     }
   }
 }
}
```
This macro is NEVER invoked.

These are the results obtained by running the above program (first two lines added for readability). A visualization can be found in figure 2.

201 *MasPar Communication Measurement Results*[201] ≡



These pictures show how long it takes on the MP-1 to send or fetch a 1-byte packet (latency time, left picture) and 1 additional byte in a large packet (right picture), depending on the size size of the machine segments, when each processor sends/fetches one packet to/from a random target processor in the same nice-rectangular machine segment.

Figure 2: **Latency and asymptotic time consumption of MP-1 router communication**

```
{
Segmentsize  Latency      time per byte
  n  log n  send fetch   send   fetch
    1  0   10967 10849    915     918
    2  1   11030 10904    921     923
    4  2   12051 11902   1011    1014
    8  3   13138 12969   1108    1111
   16  4   13817 13636   1168    1171
   32  5   15950 15728   1358    1362
   64  6   17194 16949   1468    1472
  128  7   17158 16920   1465    1469
  256  8   18392 18135   1575    1579
  512  9   16011 15789   1363    1367
 1024 10   15857 15642   1349    1353
 2048 11   16043 15826   1366    1370
 4096 12   15763 15544   1341    1345
 8192 13   15517 15312   1319    1323
16384 14   15301 15099   1300    1304
}
```

This macro is NEVER invoked.

As we see, both latency and time per byte vary with the segment size (in a weird, difficult-to-explain fashion). Both, however, vary in almost exactly the same pattern so that the quotient L_{equiv} of latency and time per byte is almost constant for all segment sizes and is also almost the same for both sending and fetching. This quotient L_{equiv} could be called the *latency equivalent*, because it answers the question "How many bytes to transfer is one latency worth?"; it can directly be used to decide which gaps to transfer and at which to start a new communication operation: all gaps of not more than L_{equiv} bytes should be transfered. From the above data, L_{equiv} can be computed to be in the range 11.5 to 12. We thus decide to tell the module to aggregate all packets that have gaps of not more than 12 bytes between them, and happily discard the thought to optimize this gap length based on the segment size, which is not statically known. A second factor besides segment size also influences latency and bandwith: The amount of load (traffic) in the network, expressed as percentage of PEs participating. The above considerations hold for 100 percent load. Additional experiments show that L_{equiv} gets higher when the load is reduced. For example it is about 13.5 for 20 percent load and about 20 for 1 percent load. Since for CuPit programs the load will usually be high, we can allow ourselves to ignore this fact and just stay with the value of 12.

These code templates are used by the module:

202 **usage.ptg**[202] ≡

```
{
 remoteFetchCode:
    [IndentNewLine] "ps_rfetch (_ME->_oe.pe, ((plural char* plural)_ME->_oe.a"
                    " + " $1 "),"
    [IndentNewLine] "            ((plural char*)ME + " $1 "), " $2 ");"
 remoteSendCode:
    [IndentNewLine] "sp_rsend (_ME->_oe.pe, ((plural char*)ME + " $1 "),"
    [IndentNewLine] "           ((plural char* plural)_ME->_oe.a + " $1 "), "
                               $2 ");"
 hideLatency:
    [IndentNewLine] "unsigned _ticksbefore_ = dpuTimerTicks2();"
                    $1
    [IndentNewLine] "timerUnuseTicks (dpuTimerTicks2()-_ticksbefore_-700);"
 spendTime:
    [IndentNewLine] "_spendTicks (" $1 ");"
}
```

This macro is attached to an output file.

The implementation of the module consists of four parts: (1) Some auxiliary procedures to handle the pairs of offset and size that represent the elements to fetch or send, (2) the procedure that computes the communication aggregation, (3) the procedure that generates the fetch code, and (4) the procedure that generates the send code. Algorithmically all these operations are quite inefficient, but we don't care since almost always the problem size is very small (below 10).

203 **remoteconcomm.c**[203] ≡

```
{
 #include "remoteconcomm.h"
 #include "pdl_gen.h"  /* Property manipulation functions and property types */

 Offset/Size Block Handling[204]
 #define L_equiv 12
 Communication Aggregation[205]
 Send/Fetch Code Construction[206]
}
```

This macro is attached to an output file.

The first thing the code generation procedures do is to convert the definition table key set to an ordered array of pairs. Each of these pairs describes the element indicated by one definition table key by its offset in the connection object and its size. The pairs are ordered by increasing offset. To to this, a number of auxiliary procedures and macros and an array of integers are defined that support the conversion. These operations number the pairs from 0 on.

204 *Offset/Size Block Handling*[204] ≡

```
{
 #define maxpairs 100
 static int os[2*maxpairs];  /* storage for pairs of offset and size */
 #define offset(i) (os[2*(i)])
 #define size(i) (os[2*(i)+1])
 static int pairsN;

 static void free_entry (int i)
 {
   /* makes room for a new entry at position i by shifting
      entries i to (pairsN-1) one position up.
   */
   int j;
```

```
  _assert (i >= 0 && i <= pairsN);
  for (j = pairsN-1; j >= i; j--) {
    offset(j+1) = offset(j);
    size(j+1)   = size(j);
  }
  pairsN++;
  _assert (pairsN <= maxpairs);
}

static void delete_entry (int i)
{
  /* deletes the entry at position i and shifts
     entries (i+1) to (pairsN-1) one position down.
  */
  int j;
  _assert (i >= 0 && i < pairsN);
  for (j = i; j < pairsN; j++) {
    offset(j) = offset(j+1);
    size(j)   = size(j+1);
  }
  pairsN--;
  _assert (pairsN >= 0);
}

static void insert_entry (int off, int sz)
{
  /* inserts a new pair of offset and at the correct place
  */
  int i;
  for (i = 0; i < pairsN; i++) {
    _assert (i == 0 || offset(i-1) < off);
    if (offset(i) > off) {
      free_entry (i);
      offset(i) = off;
      size(i) = sz;
      return;
    }
  }
  /* off is larger than any currently known offset: insert at end */
  offset(pairsN) = off;
  size(pairsN) = sz;
  pairsN++;
  _assert (pairsN <= maxpairs);
}

static void combine_entries (int i)
{
  /* combines entries i and (i+1) into entry i
  */
  int firstafter = offset(i+1)+size(i+1); /* first address after entry (i+1) */
  delete_entry (i+1);
  size(i) = firstafter - offset(i);
}

static int gapsize (int i)
```

```
{
  /* returns the size of the gap between
     end of entry i and begin of entry (i+1)
  */
  return (offset(i+1) - offset(i) - size(i));
}
}
```

This macro is invoked in definition 203.

With these basics, we can formulate a procedure that takes a may-be-used set as input and massages it into the offset/size pair representation with the optimal aggregation of elements into communication packets. Thus this procedure embodies the optimization strategy for the remote connection data communication optimization. Here is it:

205 *Communication Aggregation*[205] ≡

```
{
static void aggregateCommunication (DefTblKeySet elements, Bool nonoptimal)
{
  /* if nonoptimal is true, we don't actually aggregate, just sort. */
  DefTableKey key;
  int         i, j;
  int         off, sz;
  pairsN = 0;  /* reset packet list */
  DSiterate (elements);
  while ((key = DSnext()) != NoKey) {
    off = GetOffset (key, 0);
    sz  = GetSize (GetType (key, NoKey), 0);
    insert_entry (off, sz);
#ifdef debug
    fprintf (stderr, "%s@%d[%d]  ", SymString (GetSym (key, NoSym)), off, sz);
#endif
  }
#ifdef debug
  fprintf (stderr, "-->\n");
  for (j = 0; j <= pairsN-1; j++)  /* for all pairs */
    fprintf (stderr, "%d/%d ", offset(j), size(j));
#endif
  for (i = 0; i < pairsN-1; ) {  /* for all pairs but the last */
    /* see whether we should combine this pair with the next */
    if (gapsize (i) <= L_equiv && !nonoptimal)
      combine_entries (i);
    else
      i++;
#ifdef debug
    fprintf (stderr, "--> ");
    for (j = 0; j <= pairsN-1; j++)  /* for all pairs */
       fprintf (stderr, "%d/%d ", offset(j), size(j));
#endif
  }
#ifdef debug
  fprintf (stderr, "\n");
#endif
  /* now the offset(i)/size(i) pairs for i in 0...(pairsN-1) represent
     the optimal communication aggregation for 'elements'
  */
#undef debug
```

```
  }
  }
```

This macro is invoked in definition 203.

These operations are now used to formulate the code generation for remote connection data fetching. After calling `aggregateCommunication`, we do an ugly thing: We access the offset and size values directly to produce the `rfetch` calls.

Send/Fetch Code Construction[206] ≡ 206

```
{
  #define artificial_latency 20000  /* ticks */
  static PTGNode FetchCode;
  static int     FetchCost;

  void makeRemoteFetchCode (DefTblKeySet read, Bool nonoptimal, DefTableKey type,
                            Bool hidelatency, Bool highlatency)
  {
    int i;
    FetchCode = PTGNULL;
    if (nonoptimal && type != NoKey) {  /* fetch always the WHOLE con object */
      int Size = GetSize(type, 0);
      FetchCode = PTGremoteFetchCode (PTGInt (0), PTGInt (Size));
      FetchCost = 1600 + 80*Size;
    }
    else {
      /* nonoptimal means fetch all elements individually */
      aggregateCommunication (read, nonoptimal);
      FetchCost = 0;
      for (i = 0; i < pairsN; i++) {
        FetchCode = PTGSeq (FetchCode, PTGremoteFetchCode (PTGInt (offset(i)),
                                                           PTGInt (size(i))));
        FetchCost += 1600 + 80*size(i);
      }
    }
    if (highlatency) {
      FetchCode = PTGSeq (FetchCode,
                          PTGspendTime (PTGInt (artificial_latency)));
      FetchCost += artificial_latency;
    }
    if (hidelatency) {
      FetchCode = PTGhideLatency (FetchCode);
      FetchCost = 0;
    }
  }

  PTGNode getRemoteFetchCode ()
  {
    return (FetchCode);
  }
}
```

This macro is defined in definitions 206, 207, and 208.
This macro is invoked in definition 203.

The generation of send code is analogous to the generation of fetch code above:

Send/Fetch Code Construction[207] ≡ 207

```
{
```

```
static PTGNode SendCode;
static int     SendCost;

void makeRemoteSendCode (DefTblKeySet write, Bool nonoptimal, DefTableKey type,
                         Bool hidelatency, Bool highlatency)
{
  int i;
  SendCode = PTGNULL;
  if (nonoptimal && type != NoKey) {
    int Size = GetSize(type, 0);
    SendCode = PTGremoteSendCode (PTGInt (0), PTGInt (Size));
    SendCost = 1600 + 80*Size;
  }
  else {
    aggregateCommunication (write, nonoptimal);
    SendCost = 0;
    for (i = 0; i < pairsN; i++) {
      SendCode = PTGSeq (SendCode, PTGremoteSendCode (PTGInt (offset(i)),
                                                      PTGInt (size(i))));
      SendCost += 1600 + 80*size(i);
    }
  }
  if (highlatency) {
    SendCode = PTGSeq (SendCode,
                       PTGspendTime (PTGInt (artificial_latency)));
    SendCost += artificial_latency;
  }
  if (hidelatency) {
    SendCode = PTGhideLatency (SendCode);
    SendCost = 0;
  }
}

PTGNode getRemoteSendCode ()
{
  return (SendCode);
}
}
```

This macro is defined in definitions 206, 207, and 208.
This macro is invoked in definition 203.

Finally, we need a function that returns that part of the cost of the send and fetch operations, that does not scale with the amount of traffic in the communication network. This value is needed for the work measurement of the compiler. This fraction of the total cost is the run time (in ticks) that is to be expected when there is minimal traffic in the network. Just like the other two `get` operations of this module, `getRemoteCommCost` also relies on the values of internal variables set upon execution of the `makeRemoteXCode` procedures.

208 *Send/Fetch Code Construction*[208] $\equiv$

```
{
PTGNode getRemoteCommCost ()
{
  return (PTGInt (FetchCost + SendCost));
}
}
```

This macro is defined in definitions 206, 207, and 208.
This macro is invoked in definition 203.

PART III: Code Generation A — Introduction and Code-Templates

In this part I will describe in prose what the optimization goals of the compiler are and how they are achived. Rules for the data distribution and the name generation are given.

I also define a few header files that will be used by the MPL code generated by the CuPit compiler plus, more important, a number of large code generation templates in the form of include files, which contain generic procedures.

26 Code generation strategy

The next few sections serve to give a general introduction into the design of the code generation for the MPL CuPit compiler. They will describe what the overall optimization goals are at which the code generation aims (and what, on the other hand, it is not concerned about), what assumptions on the behavior of CuPit programs are used (and why), and what techniques are employed in order to achieve the goals (based on the assumptions). In subsequent sections, rough descriptions are given of how these techniques work.

26.1 Optimization Goals

(Introductory remarks: It turns out that most of this section says what are *not* optimization goals of the compiler. So please use the Sherlock Holmes way of thinking — what remains after all wrong has been eliminated is the searched-for truth. The now following thoughts apply to massively parallel machines with physically distributed memory attached to the individual processors where accesses to local memory is many (about 10 to 10000) times faster than accesses to remote memory.)

The optimizations implemented in the MPL CuPit compiler do not cover any of the aspects usually targeted by optimizations in modern compilers for sequential machines, such as register allocation, peephole optimization, automatic inlining, instruction scheduling, etc. Since our compiler translates into MPL, we can leave most of these optimizations to the MPL compiler; the others are simply ignored for requiring too much effort compared to the negligible scientific benefit they have in the context of the study the compiler was created for (see section 1).

The optimizations do also not cover many of the aspects targeted by modern compilers for parallel machines, such as automatic parallelization, vectorization, elimination of synchronization points, or overlapping communication with computation. Some of these optimizations are not applicable at all to CuPit, some are not applicable on the MasPar, some are of little or no use on the Maspar, and others are again ignored.

> *The main optimization goals of the MPL CuPit compiler are (1) intra-object locality and (2) static load balancing.*

Intra-object locality means that whenever an operation is performed that accesses only the data of a single compound data object (such as a connection or a node), all this data is available at the processor performing the operation without any interprocessor communication. As we will see, this requirement can easily always be fulfilled for nodes. For connections, however, it can only be fulfilled in roughly half of the cases: Connection operations can be called from either of the nodes at their two ends (the node, where they are attached to an input interface, and the node where they are attached to an output interface), but their actual data can only be allocated at one of the two ends. Thus only parts of the connection operations can maintain locality. However, a correct decision as to at which end of a connection to store the connection's data guarantees that at least half of the work can be performed locally.

Static load balancing means to achieve load balancing with only static work distribution decisions. Load balancing means that in any parallel section as many processors should do useful work as the parallism inherent in the problem allows. Or, put the other way round: As few processors as possible should wait for others at any time during the program run while these others perform work they could have shared with the waiting ones. Static work distribution decisions means that whenever a parallel work section is started during the program run, the decision which processors perform which part of the work is fixed. "Static" does *not* mean the decisions have to be made at compile-time; they may be computed immediately before the actual work begins — they only must not change during the execution of any single parallel work section. The rationale for this definition lies in the maintenance of data locality: Completely dynamic work distribution is inherently unable to guarantee data locality (unless you move the data along with the work, but that is just as expensive as non-local data access). Work distribution in the "static" sense above, though, can exploit data locality if the data distribution was chosen in anticipation of the work distribution decision. This means that static load balancing is able to maintain data locality if the work distribution decisions do not change too often. In the CuPit compiler, we take the inverse approach: We guarantee data locality a priori; all work distribution decisions are made upon data distribution time and data distribution does not change too often.

The realization of these optimization goals depends on certain assumptions which are discussed in the next section.

26.2 Assumptions

The techniques used to achive the above optimization goals are based on a set of assumptions that are justified by the properties of the application domain "neural algorithms". The following assumptions are used:

Assumption 1: For any one parallel call of a connection operation, the work to be performed is roughly the same for all participating connections. *Utility:* This assumption allows to relinquish load balancing efforts on a sub-connection level. It also allows to use the number of connections as a measure of work size within any single call. *Justification:* In neural algorithms, connections are usually very primitive objects. The operations performed on them hardly ever involve loops at all. Without loops, however, the work to be done will not differ too much between different connections.

Assumption 2: For any one parallel call of a node operation, the work to be performed is roughly the same for all participating nodes, i.e., work per node (except for connection work) is constant within one node group within any one call. *Utility:* With this assumption we can limit load balancing efforts on the node level to connection work balancing. *Justification:* To justify this assumption, we basically use the same argumentation as for the connections above. But the fact that the work needed for the connections is embedded in the work for nodes makes the situation more complicated: Should some of the nodes inactivate themselves for most of the operation, the assumption does not hold. However, if such inactivation decisions are data-driven and cannot be anticipated individually by the programmer, it is hardly possible to achieve load-balancing with *any* static method. Thus the assumption is acceptable, although we recognize that it may be a significant simplification for certain programs.

Assumption 3: The data distribution has to be changed not too often, i.e., the work needed for computing a new data distribution and executing the reallocation accounts for only a small fraction of the overall run time. *Utility:* Rare data distribution change allows to invest a lot of resources into doing it cleverly, since the amortization period for such resource investment is long. *Justification:* In neural algorithms, network topology changes usually occur only after one or even several epochs of training. Assuming that the amounts of training data are not too small, topology change will occur not too often. Due to the next assumption below, though, data distribution changes are usually necessary only during network topology changes.

Assumption 4: The behavior of the algorithm in respect to the load at individual data objects changes only smoothly (if at all). *Utility:* Smooth evolution of behavior means that decisions based on data from the execution history that were good when they were made degrade only slowly. In particular, the load balancing provided by a certain data distribution will not get very bad from one moment to the next (unless the topology is changed). *Justification:* Neural algorithms are iterative local search processes. The behavior of the algorithm itself does usually not change much during the whole learning process, i.e., almost the same program code is used all the time. The behavior of the data (the network) only sometimes changes erratically in a well-balanced neural algorithm; most of the time the evolution is almost continuous.

Assumption 5: Trying to optimize extra-object locality does not pay off. Extra-object locality means that nodes which communicate with each other (i.e. have a connection and use it) are allocated on the same processor (are "co-located"). *Utility:* Not optimizing for co-location saves a complicated connectivity analysis and allows to design the data distribution for other efficiency-enhancing properties that could not be achieved when co-locality was required. *Justification:* The validity of this assumption strongly depends on two factors: The target machine and the topology of the neural networks to train. As far as the machine is concerned, the assumption holds on the MasPar, because (1) the MasPar is very fine grained, so that the number of co-local nodes is very small anyway, compared to the number of non-local nodes, (2) communication cost on the MasPar is relatively well-balanced compared to computation cost, (3) communication cost does hardly depend on distance. As to how much co-locality not utilized by a random distribution is available in a network topology is very difficult to say.

26.3 Techniques

Locality is achieved by using a class of data distributions that guarantees local node operations in all cases and local connection operations in a part of the cases (as given by the at-which-end-should-the-data-be dilemma, see below). The central idea of this data distribution is to allocate not only one processor per node but a whole processor block. The node is replicated across this block and the connections attached to the node are distributed across it.

Load balance is achieved by adjusting the parameters of this data distribution cleverly. The most important aspect of this parameter optimization is to allocate a number of processors for each node that is proportional to the connection work load of the node.

The next three sections will describe the basic ideas of the data distribution and the data distribution parameter adjustment.

27 The code generation types

This compiler can generate code in three different versions.

1. A *plain* version . This version has data locality but does not perform any load balancing.
2. A *statically optimized* version . This version is statically optimized solely based on the analysis of the program source text and a machine model. It performs load balancing based on the assumption that the work performed is the same for each connection at each interface.
3. A *run time information collection* version (or short *rti* version). The rti version contains additional code to measure the actual run time of connection operations. Based on these measurements, the data distribution can be optimized to the actual behavior of the program instead of just a static estimation. Such optimization is done dynamically when the rti version is run.

Which of the three versions of code to generate is selected at compile time by compiler options: `-dumbbalance` selects the statically optimized version, `-nobalance` selects the plain version, by default the compiler generates the rti version. In the generated code, the preprocessor symbol `_codetype_` has the value 0 for the plain version and 1 for the run time information version.

28 The data distributions

Any data distribution used may have the following useful properties to a higher or lower degree: Locality, load balancing, direct addressability, dynamic extendability, dynamic reducability, propagation of information about dynamic changes from connections to nodes and from nodes to nets, index computation, maxindex computation.

We use two different data distributions because not all of these goals can efficiently be achieved with a single one. Topology change operations are done only in one of the forms, all others in both. The criterion for deciding when to switch from one distribution to the other is the generation of replicates: While the number of replicates is 1 (i.e. changes in topology are possible) we usually use a data distribution we call *form 0*. While there are several network replicates (and changes in network topology are not allowed), we use a data distribution we call *form A*. The major difference between the two forms is that form A achieves load balancing while form 0 gives up load balancing in order to achieve direct addressability and ease dynamic change.

In the following we will describe the two data distributions, using the following terminology: Let `net` be a network variable having `r` replicates. `nodes` is a node group of `net` consisting of `n` nodes. `con1, con2,...` are connection interfaces of a node and have `c1, c2, ...` connections attached. *Nice-rectangular* means a rectangle whose sides have lengths 2^i and 2^j, respectively, with $\bar{i} - \bar{j} \leq 1$, i.e, side lengths are powers of two and either are equal or differ by factor two.

28.1 Form A

The basic idea behind the form A data distribution is to allocate not only a single processor for each node of a node group, but a whole block of processors; the connections attached to this node are distributed accross this processor block. Replicating the node data accross the processors in the block allows for locality between a connection and the node is it attached to. Chosing the size of the node's processor block in proportion to the amount of work required by the connections results in balanced load. For replicated networks we use the same method after dividing the machine into segments, one for each replicate. Here is a somewhat more detailed description:

The processor array is segmented into `r` nice-rectangular parts (called *machine segments* or just *segments*). Either `r` is a power of two or we round it up to a power of two and do not use all allocated replicates.[3]

The exactly same data distribution is used within each machine segment. Each node group is allocated separately. For each node in a node group, we allocate a nice-rectangular subsegment (called a *processor block* or *node block* or just *block*). Processor blocks for nodes belonging to replicate a are always allocated to lie completely within the segment a. Connections are distributed accross the processors of the block in a "cycle" fashion: on a block of size b, connection i is located on processor ($i \bmod b$) within the block. The size of the processor block is chosen for each node individually so as to achieve optimal load balancing for the connection operations.

The local data of each replicate of the network variable is allocated on the processor at the upper left corner of the corresponding machine segment. The local data of each replicate of a node object is allocated on the processor at the upper left corner of the corresponding processor block. The other processors of the same segment or block all carry duplicates of the same data, called *shadow* duplicates; these duplicates have identical values at any time during the program, i.e., whenever this invariant is violated it will be reinstantiated before any other operation occurs. The local data of a connection is stored at either its input or its output end. At the other end, allocating a connection means allocating a pointer to the actual data. The decision whether to store the connection data at the input end or the output end is made for each connection type individually.

Virtualization is never necessary for network replicates, since we never allocate more network replicates than there are processors.[4] Node groups are subject to virtualization when the sum of the numbers of processors in its nodes' processor blocks exceeds the number of processors of the machine (or machine segment). For each node group, the number of nodes (primary plus shadow) allocated on one physical processor is always the same for all processors of the machine; some of these nodes may be marked as non-existing, meaning they are not actually used. Individual nodes are not subject to virtualization; a

[3] The reason for always working with powers of two (here as well as in other contexts) is that it makes many computations much more efficient, in particular address computations and boundary tests.

[4] *Justification:* Having more replicates than processors cannot achieve higher efficiency but will make data management more complicated. We can thus save this effort for sake of easy implementation without losing an important feature. Given the large amount of processors and the small amount of memory per processor available on the MasPar, it is hardly possible to use more than one replicate per processor, anyway.

processor block is always allocated on a physically contiguous block of processors. Within each virtualization layer of a node group, the number of connections attached to the same connection interface and allocated on one physical processor is always the same for all processors of the machine; some of these connections may be marked as non-existing, meaning they are not actually used. Node virtualization may either be done such that in any virtualization layer of a particular node group there are only nodes with subsequent indices, i.e., each virtualization layer contains exactly a complete subrange of the total index range of the node group (linear block arrangement) or it may be done such that any virtualization layer may contain nodes with arbitrarily mixed indices (random block arrangement). Random block arrangement has the advantage to allow minimal waste of virtual processors by optimizing the block layout globally across all virtualization layers. Linear block arrangement has the advantage that it is easier to construct than an optimal random block arrangement and that it allows more efficient execution of operations on slices of the node group when the virtualization factor is large. Since the maximum waste of virtual processors can be limited by restricting the fraction of a segment occupied by the largest node block, the compiler uses linear block ordering.

28.2 Form 0

The main differences to form A is (1) that the processor blocks of each node have the same size and are always arranged linearly across the PE array and (2) that there is always only a single network replicate. Property (1) destroys load balance if different amounts of work are required at different nodes, but it also has advantages: It is possible to address the upper left corner of any node block directly. We can (and do) always work without node level virtualization and we can preallocate room for dynamic extension of node groups. Direct addressing is very valuable for easy formulation of data structure modifications (addition and deletion of nodes or connections). The connections are allocated accross any node block in the same fashion as in form A, still in no particular order. The block size is still a power of two.

29 Data distribution parameter decisions

29.1 The run time measurements

The individual data distribution parameter decisions described below are based on empirical (i.e. run-time) data about the behavior of the program. The only type of cost considered is run-time. The compiler uses the following basic measures:

1. measured per node group, per connection interface: total work (i.e., effective run-time) at each connection interface. Purpose: used to compute relative node block sizes from numbers of connections at each interface. Caveat: This value depends on whether connection data is remote or not. The measurement thus interferes with measurements trying to decide whether connection objects should be placed at input or at output interfaces.
2. measured per node group, per connection interface, per replication/non-replication phase: total work (i.e., effective run-time) at each connection interface. Purpose: auxiliary measurement to compute the correct information for measurement 1, which is possible only indirectly, because phases during which topology changes can be made (and thus the data distribution has unoptimized form) have to be taken out of account.
3. measured per node, per connection interface, per replication phase: total number of connection operations called. Purpose: compute absolute node block sizes necessary for good load balance despite parial non-activity in the nodes. (The optimization that uses this second measurement and the measurement itself is not implemented completely.)

29.2 Where to put connection data

The actual data portion of a connection can be put at either its input or its output end. The other end contains just a pointer. This decision can be made for each connection type independently and is stored in the `Dataloc` property of the connection type. Which way is best depends on the distribution of connection operations.

To make the decision, the code generation could implement measurement code to compute the cost of both versions at once for the test run. During the test run the total cost of each alternative is measured and the decision is made accordingly in the generation of the final code. Computing the cost of the alternative would work in the following fashion: During each form A phase, we collect for each connection procedure call the total communication cost and the total computation cost. The communication cost (time used for fetching and sending remote data) is computed for the data end as well as for the remote end of the connection by inserting a dummy communication operation at the data end. These costs are then accumulated per connection type over all connection procedure calls that happened in the test run for connections of this connection type, giving total communication costs for the data end and for the remote end ($Comm_d$, $Comm_r$) and total computation costs for the data end and for the remote end ($Comp_d$, $Comp_r$). The data end should be changed to the remote end, if

$$Comm_d + Comp_d + Comp_r < Comm_r + Comp_d + Comp_r$$

or kept where it is otherwise.

The problem with this approach is that it interferes severely with node block size optimization: When the numbers of connections at different interfaces of a node differ significantly, the node block size will change dramatically depending on which interfaces are data ends and which are remote ends. Different node block sizes however, will result in different degrees of virtualization and this, in turn, will change the value of total communication costs. This problem stems from the fact that we need to know the absolute costs, not the relative per-connection cost. Absolute costs however, can be known only after node block size decisions are made.

Because of this problem, this compiler does not implement an automatic optimization of connection data localization. Instead, a compile-time switch (`-conatout`) can be used to change the localization decision by hand; the user must try both variants and use the one that is better. Since most programs have only one connection type, the compiler switch changes the localization of all connection types at once.

29.3 How to fetch remote connection data

A second problem resulting from the fact that connection data can be used at two addresses but can be allocated only at one is the following: What is the most efficient way to access remote connection data in a connection operation[5] called for a remote connection ?

There are several different methods: (1) communicate exactly what is needed exactly when it is needed on a per-element basis, i.e., communicate for each individual element access. (2) dito, but use cacheing to optimize multiple reads and/or write-behind to optimize multiple writes. (3) at the beginning of the operation, fetch any element that might be read during the operation and at the end of the operation send any element that might have been written to during the operation; the set of elements possibly read or written is determined statically. (4) at the beginning of the operation fetch the whole connection object if *any* element may be read and at the end of the operation send back the whole connection object if *any* element may have changed.

Method (1) will be very inefficient if some elements are accessed often within one connection operation. Method (2) is prohibitively expensive on the MasPar because it has a large management overhead which can not be amortized on the MasPar since communication is relatively efficient and the objects are rather small. Method (3) will be inefficient if the fraction of elements used is large and their number is not negligible, because the constant efforts for each fetch or send operation involved will sum up to exceed the time saved by not fetching or sending unused elements. Packing the used fields into a single communication packet is also not feasible due to the overlarge management cost compared to object size. Method (4) will be inefficient if only a small fraction of the connection object is used. Of course the method can be selected independently for reading and writing: we might want to chose (3) for reading but (4) for writing.

[5] A connection operation is one call of a connection procedure from a node procedure

The approach taken by the compiler is a modified version of method (3). Type analysis computes the may-be-read and may-be-written set of elements, along with their size and offset. A static cost model of bandwith and latency then computes which of these elements should be fetched or sent individually and which should be aggregated into larger communication packets. Such larger packets may fetch or send superfluous elements if this is cheaper than starting a separate communication operation.

Note that the analysis treats each element as atomic, i.e., record and array elements are always fetched or sent as a whole.

29.4 Decision about node block size

The selection of node block sizes is done for each node group separately and has two different aspects: (1) The relative size of the node blocks is responsible for load balancing *among* the nodes in a group. The size of each node block should be proportional to the total amount of work that must be done for the connections attached to the node. (2) The total absolute size of node blocks is responsible for machine-wide load balancing when there are node operations that have to be executed only for a subset of the nodes in the group: The larger the node groups, the smaller is the fraction of the group that is in one virtualization layer (i.e. executes concurrently). Such small numbers of nodes executing concurrently reduce the expected amount lost computing power due to inactive processors. On the other hand, the large node blocks that are necessary to fill the machine with a small number of nodes have three disadvantages: (a) they increase the memory overhead for node replicates, (b) they reduce the degree of freedom for node block size variation to achieve relative load balancing, and (c) they make connection reductions and broadcast over the node block more expensive due to the increased total communication distance.

In this compiler, we focus on aspect (1) from above. The absolute size of the blocks (aspect (2)) is always chosen so as to fill the machine exactly once (or, in certain cases, twice). The compiler can find the optimal relative size of the node blocks if it has the following knowledge: At the end of a form 0 phase, it knows for each interface of each node in each node group, how much work will be done per connection at this interface on the average of the next form A phase[6]. For a practical implementation, this ideal of the available knowledge has to be reduced: We cannot really *know* the future work needed, so we have to estimate a prediction based on the previous phases, if any.

Given these restrictions, we can dynamically collect information during form A phases in order to make good decisions on block sizes for the next form A phase. An additional restriction is imposed in our implementation: All node blocks have sizes that are powers of two. Thus, relative load balancing is imperfect within a bandwith of factor two. This decision was made because many computations get so much simpler when block sizes are powers of two (in particular all address computations).

29.5 Decision about node virtualization

Given the above considerations, one important open question is how to handle new nodes. There are two ways to handle nodes that were created during the last form 0 phase. (1) We can just treat them like any other. This means to assume they will need the same work per connection as the older nodes at each interface. Under this assumption it is sensible to put all nodes in one virtualization layer (except when the number of connections at some nodes is so large that this reduces the load balancing capabilities too much).

The alternative is (2) to assume that all new nodes will behave the same but not necessarily the same as the old nodes[7]. Under this assumption, it would be clever to have one (or several) virtualization layers that contain only the new nodes and one (or several) others containing the old nodes. Block sizes for the old nodes should be chosen based on the already known work per connection values of the interfaces, while block sizes on the new nodes should be chosen based on the assumption that connections at all interfaces need the same amount of work.

[6] Remember that we assume that the work for each connection is the same for all connections at the same interface of the same node.

[7] This assumption is true for algorithms that train only or mostly these new nodes in the next phase.

In this compiler, we select the one of these assumptions that seems to have higher validity during the test run: If the new nodes regularly have much higher (or much lower) work per connection than the old nodes, we prefer assumption (2), otherwise we use assumption (1). The decision is made on a per group basis. Not implemented completely.

29.6 Decision about number of network replicates

Using a larger number of replicates has the following advantages: (1) the segment of each replicate is smaller, reducing the non-locality of communication, (2) the number of different nodes active at the same time tends to decrease, possibly enhancing load balance if not all nodes are active during all of a node operation and increasing machine utilization if the absolute number of active nodes is small, (3) the number of corresponding nodes active at the same time increases, enhancing the machine usage for winner-takes-all statements (this is almost the same as (2)).

The disadvantages of a larger number of replicates are: (1) on the average more replicates will be unused at the end of the epoch when the number of examples is not divisible by the number of replicates, (2) more memory is required leaving less room for training data or even overloading the machine completely, (3) replicating and merging takes longer, (4) the possible variance in node block size is lower (since the segments are smaller), perhaps hampering good load balance.

Proper decision about the number of network replicates is not implemented in this compiler. Instead, the compiler uses a very simple method to select the number of replicates: It uses the maximum power of two that is less than or equal to the maximum allowed number of replicates. Should this number be less than the minimum allowed number of replicates, the maximum allowed number is used. Since all node block sizes are powers of two, the machine can only be used completely if the number of replicates is also a power of two (because then and only then will the number of processors in the machine segment of each replicate be a number of two, as is shown in the next section).

29.7 Segment and block layout

The computation of segments is straightforward for two reasons: (1) segmentation is regular and (2) the segments always fill the machine exactly because the number of segments, the size of each segment, and the size of the machine are all powers of two (and segments are always nice-rectangular). The same statement is true for the computation of block layout in form 0.

Form A block layout (given the required size of each block) is essentially a variant of the two-dimensional bin-packing problem with the following formulation: Pack B nice-rectangular objects into several nice-rectangular bins of identical size; use as few bins as at all possible.

The above is the formulation for optimal random block arrangement. For the linear block arrangement case, the B objects are ordered into a sequence. We pack the longest possible subsequence beginning with object 0 that fits into one bin before beginning to fill the next bin with the same method applied to the rest of the sequence.

It turns out that the nice-rectangularness of bins (virtual segments) and objects (node blocks) together makes the problem so easy that it can be solved in time proportional to $B \cdot \log_2(B)$ (while the general problem requires exponential time). The suggested algorithm is also parallelizable and works in time proportional to $\log_2^2(B)$ on B processors.

The algorithm is based on the observation that there are only $\log_2(B)$ different possible object sizes and works roughly as follows:

1. sort the blocks by their size
2. Do the following for each existing blocksize b from maximum to minimum:
3. mark all bin positions that (according to their position within the segment) might hold upper left corners of b-sized blocks with 'here', all other positions with 'no, here not'.
4. change all 'here' marks on already-occupied positions to 'no, here not'.

5. enumerate the 'here' marks, i.e., store the value $i-1$ at the ith 'here' mark. The order is through rows first, columns next, segments last.
6. allocate block number i at the position holding value i.

The sorting of blocks by size needs not really be done. Instead, it is also sufficient to enumerate the blocks of the just-to-be-positioned size just before positioning them.

29.8 Reorganization scheduling

Reorganization of the network representation (or parts of it) occurrs in the following situations:

1. When `REPLICATE network` is called.
2. When `CONNECT` is called and there it not enough room for the new connections in the relevant local connection arrays (only the affected connection arrays are reorganized).
3. When `EXTEND` is called and there are not enough free node blocks left in the current representation of the respective node group (only the affected node group and its connections are reorganized).
4. When `REPLICATE node INTO 0` is called (only the affected node group and its connections are reorganized).

This means in particular that no automatic reorganization is done after any number of `REPLICATE connection INTO 0` or `DISCONNECT` calls. If such reorganization is needed, the programmer must call `REPLICATE network INTO 1` explicitly.

30 The topology-changing operations

The next few sections give a very rough overview of the steps involved in the individual topology-changing operations. All these operations can only be applied while the network is in form 0 representation.

30.1 Connect

`CONNECT a[x...y].out TO b[v...w].in`
Count how many yet-unused connections are allocated for each node in the affected parts of groups `a` and `b`. If there is at least one node where there are not enough of them, reorganize and make enough room by allocating a new, larger connection array. Generate and initialize the connection objects and their descriptors. The number of connections stored in the interface descriptor is *not* kept current.

30.2 Disconnect

Mark the connection objects as unused (at both ends!). No reorganization is done. The number of connections stored in the interface descriptor is *not* kept current.

30.3 Extend

Extending a node group by a negative number of nodes means marking all connections of the to-be-deleted nodes and the nodes itself as non-existing. Update the node group descriptor.

Extending a node group by a positive number of nodes means: Test whether they fit into the yet-unused space in the node group. If not, reorganize and make room accordingly, i.e., compute the new block size needed to fit all nodes onto the machine without virtualization and move all nodes and their connections into these new blocks. Initialize the new nodes and update the node group descriptor.

30.4 Replicate network

There are four cases, described by the status before the replication (one replicate or many replicates) and the requested status after the replication (one replicate or (possibly) many replicates). These cases are discussed individually.

30.4.1 One into one

There are four subcases, depending on whether the old and new data distribution, respectively, are form 0 or form A: Form 0 to form 0 is just a reorganization. Form 0 to form A works like 'one into many', except that the last step (copying segment 0 into segments 1 to n) is void. Form A to form 0 is just like 'many into one', except that no merging is needed. Form A to form A is impossible because it is equivalent to 'many into many'.

30.4.2 One into many

If the "many" is an interval we first determine the optimal number of replicates from the interval. If necessary, the number is rounded to a power of two (additional replicates, if any, will be marked as non-existing). We physically always use a number of replicates that is a power of two, because this simplifies address computations and computation of boundary conditions significantly.

In the next step, the machine is segmented into replicate segments and the network is reorganized from form 0 on the whole machine into form A in segment 0. Last, segment 0 is copied onto all the other segments and the descriptors are updated accordingly.

The reorganization from form 0 into form A works roughly as follows:

1. Make a copy of the network variable: `old_net := net`.
2. For each node group `g` in the old network do the following 3 steps:
3. — decide which block size to use for each node
4. — compute block layout
5. — allocate the new nodes and connections and copy the node and connection data of the group
6. For each node group `g'` in the new network do the following step:
7. — compute the correct remote pointer of each connection

The construction of the correct remote pointers is done in two steps, the first of which is done during connection copying. Consider one connection `con1` and its corresponding remote connection `con2`. At the beginning these two objects have remote pointers `oe1` and `oe2`, respectively, that satisfy the invariant `oe1 = &con2 AND oe2 = &con1`. The goal of the pointer computation is to assert the corresponding invariant for the new connections `con3` and `con4` (corresponding to `con1` and `con2`, respectively).

In step 1 we set `oe3 := oe1; oe4 := oe2; oe1 := &con3; oe2 := &con4`, in this order, i.e., we copy the old remote pointer values into the new remote pointer objects and then reset the old remote pointers to point to the new *corresponding* connection objects. Now the old remote pointers so to say point to their new self and the new ones point to the old remote connections. In step 2 we follow these latter pointers to compute the new pointers: `oe3 := oe3->oe; oe4 := oe4->oe`. The two steps cannot be done in one, because `con1` and `con2` may (and usually will) belong to different groups and are thus not generally handled at the same time.

30.4.3 Many into one

Apply the `MERGE` procedure, compute the appropriate node block size for each node group, reorganize form A into form 0 using the whole machine to distribute the form 0 data. The reorganization can be done in a way very similar to that used for the reorganization from form 0 into form A and is in fact implemented in the same procedure.

30.4.4 Many into many

This replication is not allowed according to the language definition. Instead, the programmer must explicitly write a replication into one first.

30.5 Replicate node

Replicating a node into several nodes is not implemented. It is quite difficult to implement, because the connections have to be cloned. Replicating connections is not easy because it requires adding connections at possibly many foreign node groups at once (and must avoid cloning intra-group connections twice).

Replicating a node into null replicates means marking all its connections and the node itself as non-existing and then reorganizing the node group with new indices computed for the nodes.

30.6 Replicate connection

Replicating a connection into several connections is not allowed by the language definition, because connections must be identifiable uniquely by the pair of interfaces they are attached to.

Replicating a connection into null works much like a `DISCONNECT`: The connections are just marked as non-existing.

31 Name generation rules

Since the generated code is MPL source code, a lot of MPL identifiers have to be generated. The scoping rules of CuPit were chosen to conform with those of MPL, so no special treatment is necessary for names that directly correspond to names in the CuPit program.[8] However, a lot of additional names are needed, in particular because many CuPit objects are implemented as a multitude of objects in the MPL program (for instance a parallel object procedure as a simple plural procedure and as a virtualized plural procedure). A second reason is that we need names for some objects that have no name in a CuPit program (e.g. the MERGE procedures).

The names of these objects are created systematically using the rules described in this section.

The basic technique to avoid name clashes between a name that was generated from a CuPit name A (or from nothing) and a CuPit name B is to use the underscore character as a separator between the parts of a generated name. Since the underscore is not allowed in CuPit identifiers, this technique will always generate a name that cannot be a plain CuPit name.

31.1 Type definitions

Type names are used directly in the generated program. For connection, node, and network types, there is a second version of the type that contains only the local data of the type (i.e. no connection interfaces for nodes, no nodes for networks, and no descriptors in any case) and is named by appending `_0` to the type name.

31.2 Named procedures and functions

The names of global procedures and functions are used directly in the generated program for the singular version of the subroutine. For the plural version we prepend `p_`. This is the same naming convention

[8] Note that this way no identifiers may be used in a CuPit program that are keywords in MPL. See appendix B

that the MPL library uses. A plural version is generated only for free subroutines, not for those in the central agent, because these cannot be called in a plural context.

`REDUCTION` and `WTA` functions are always plural; their names always have `p_` prepended.

Since the same name can be used in CuPit for a global and an object subroutine or for two object subroutines in different types, we append the name of the type to the subroutine name of an object subroutine, i.e. use the form *subroutinename_typename*. Although object subroutines are always plural, we do not prepend a `p_` prefix here. For object procedures, though, we need an additional virtualized version of the procedure. This version is named by prepending an `a_` (Mnemonic: `all_`).

31.3 Unnamed procedures, operators, and builtin objects

Unnamed procedures are the initialization procedures for structured types, the procedures that implement the input and output assignment operators, and the `MERGE` procedures. (Unnamed means that no name for them is mentioned in the CuPit program.) They have names of the form `INIT_`*typename*, `INPUT_`*typename*, `OUTPUT_`*typename*, and `MERGE_`*typename*, respectively. While the `INIT_`, `INPUT_`, and `OUTPUT_` procedures already include all virtualization activities, there is an additional wrapper needed for the `MERGE_` procedures. This wrapper is named `a_MERGE_`*typename*.

The names of the operators are the same as used in the operator identification process (see section 22.1 on page 103ff), unless the operator has an equivalent in MPL in which case this equivalent is used directly. `ME` and `YOU` are used as object names in the MPL program directly. For the names of the run time system procedures and functions see part "Run Time System".

31.4 Structure elements and desriptors

The names of elements of record, connection, node, and network types are used directly. This is possible, because each `struct` has its own scope in MPL. The names of descriptors that correspond to some element `x` have the form `x_D`.

Within the descriptors, the following naming conventions hold: Elements holding integral numbers that are always powers of two and are represented in logarithmic form have names that begin with `l`. Elements representing ordinal numbers (indices, always beginning at zero) have names ending in `I`. Elements representing cardinal numbers have names ending in `N`. These conventions apply to many local variables used in the generated code as well.

31.5 Data objects

The names of global and local data objects of the CuPit program are used directly. Data objects introduced by the compiler are guarded against name clashes by always choosing names that begin with an underscore character. The rules for `l` prefix and for `I` or `N` suffix described above also apply.

31.6 Run time information

Parts of the generated code and data structures are not necessary to implement the semantics of the CuPit program but have the purpose to collect run time information. As far as the above rules are concerned, this run time information collection code (short: rti code) is also structured like ordinary code. To suppress the rti code when it is not needed, two different approaches are used at different points in the code generation: At some places, the compiler generates different code when it is in rti code generation mode than when it is not. At other places, the rti code only needs to be omitted and this is achieved by enclosing it in an `#if` section.

32 Data types

We need a number of data types to be used in the generated code: The builtin types of CuPit, the data structure descriptors, and a few auxiliary types. These types are defined in two header files that are given in the next sections: `cupittypes.h` contains the MPL definitions for the builtin types of CuPit, `descriptors.h` contains the descriptor and auxiliary types.

32.1 CuPit builtin types

Well, there is not much to say about these declarations, so here we go:

lib/cupittypes.h[209] ≡ 209

```
{/*
 File: cupittypes.h, typedefs for CuPit builtin types
 RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 #ifndef CUPITTYPES_H
 #define CUPITTYPES_H
 typedef unsigned char    Bool;
 typedef int              Int;
 typedef short            Int2;
 typedef char             Int1;
 typedef float            Real;
 typedef char*            String;

 typedef struct { Int  min, max; }  Interval;
 typedef struct { Int2 min, max; }  Interval2;
 typedef struct { Int1 min, max; }  Interval1;
 typedef struct { Real min, max; }  Realerval;

 #define true    ((char)1)
 #define false   ((char)0)
 #define VAR
 #define CONST   const

 #endif
 }
```

This macro is attached to an output file.

32.2 Descriptor types

The descriptor types carry all administrative information related to the individual parts of a network, i.e., the network and its replicates (`_network_D`), the node arrays and node groups (`_node_group_D`), the individual nodes in each replicate and their shadows (`_node_D`), the connection interfaces (`_interface_D`) at each node, local connections (`_connection_D`), and remote connections (`_remote_connection`). In fact, `_remote_connection` is not really a descriptor type, because there is no data object that it describes; all remote connections of whichever connection type are represented solely by a `_remote_connection` object.

The tasks that must be solved by the descriptors are the following:

1. Address computation for the standard object procedure calls, reductions, and winner-takes-all statements.
2. Address computation and memory management guiding for topology change operations and data distribution reorganizations.

3. Storing of the run time information collected by the rti code.

The term "address computation" is used loosely here and includes activity checks and the computation of iteration spaces (boundaries, index ranges, etc.).

All descriptors are allocated as plural, i.e., on each processor. Some of the information in a descriptor may be a priori identical on many or even all processors; replicating it, though, makes the information available locally, simplifying code generation and improving efficiency. The memory overhead induced by such replication is practically irrelevant unless the values are *always* equal on *all* processors: Since we are unable to balance memory consumption across the processors, the maximum of the memory needs at any single processor determines the effective overall memory consumption and would not sink if the replication was not made, because in the chosen data distribution scheme at least processor 0 has to hold a copy of each descriptor. Some of the descriptor data, though is guaranteed to be the same on each processor in any case. Such data could be represented by singular data objects, we call it *quasi-singular*. The reason for not actually implementing it as singular data is that the management of descriptor data would then become considerably more complicated in the compiler: Since MPL does not allow to mix singular and plural components in a single struct, we had to carry two data objects (one singular and one plural) all through the code generation instead of only one that we need now.

Descriptors may carry the following information, although not all of this information is present in each descriptor type. The descriptor of an object A may contain:

1. Self-existence indicator (`exists`)
2. Self-identification of A (`meI`)
3. Description of the part of the processor field used for sub-objects of A
4. Degree of virtualization (`localsizeN`, `con_ls`)
5. A backpointer to the descriptor of the object A is a part of (`boss`)
6. Run time information about the usage of A

Descriptors so to say answer the questions "Am I ?", "Who am I ?", "Where are my subordinates ?", "Where is my boss ?", "What am I doing ?". Note that this compiler is a world in which "Am I ?" is not a yes/no question due to shadow nodes.

Now follow the actual data structure declarations for the descriptor types. Note that the ordering of the elements is chosen so as to minimize the total size of the type by avoiding alignment gaps as much as possible.

210 **lib/descriptors.h**[210] ≡

```
{/*
 File: descriptors.h, type definitions for cupit data structure descriptors
 RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 #ifndef DESCRIPTORS_H
 #define DESCRIPTORS_H

 Machine Descriptors[222]
 Auxiliary Types[217]
 Network Descriptor Type[211]
 Node Group Descriptor Type[212]
 Node Descriptor Type[213]
 Interface Descriptor Type[214]
 Connection Descriptor Type[215]
 Remote Connection Type[216]

 #endif
}
```

This macro is attached to an output file.

Exactly one network descriptor is allocated on each processor for each network used. The network descriptors carry mostly quasi-singular information: `formA`, `repN`, `lrepN`, `lxN`, and `lyN` are guaranteed to be the same on all processors. Note that `repN` is the logical number of replicates while `lrepN` is the physical number (the `l` means logarithmic, not logical); actually, `lrepN` is redundant, because it always equals `lprocN - (lxN+lyN)`.

Network Descriptor Type[211] ≡ 211

```
{
 /* one per processor per network */
 typedef struct {
   _realness exists; /* Whether this Network replicate is actually used or not */
   _bool formA;      /* Whether the network is currently in formA or form0 */
   _sint meI;        /* My own replicate number */
   _sint repN;       /* number of logical replicates of network */
   _bint lrepN;      /* log number of physical replicates of network */
   _bint lxN, lyN;   /* logarithm of size of my machine segment */
 } _network_D;
}
```

This macro is invoked in definition 210.

One node group descriptor is allocated on each processor for each node group in each network used. All node group descriptor data is quasi-singular: For `nodesN` this is necessarily so, because all these values in a plural representation refer to the number of nodes in the same node group. For `localsizeN` and `boss` the values are the same since all dynamic data allocation is done with `p_malloc`; this procedure always allocates the same amount of memory on each processor at the same address on each processor. `better2virt` is used to accumulate evidence points for or against the decision to use two virtualization layers for this node group (one for the 'old' nodes and one for the 'new' nodes) as opposed to using just one. These evidence points are collected during the rti version run and are used for the decision in the optimized code version. One evidence point is added for each formA phase with old and new nodes in which the new nodes have significantly more or less work per connection than the old nodes, and one evidence point is subtracted for each such formA phase, where this is not the case. Two virtualization layers are used for the node group in the optimized code when at the end of the rti version run the number of evidence points as stored in `better2virt` is strictly positive, and one layer otherwise. The value is valid only in replicate 0. `newnodesN` is the number of nodes that are currently considered *new*; these are always the nodes with the largest indices. New nodes without old nodes are no newnodes! Node groups always exist, so no `exists` indicator is needed.

Node Group Descriptor Type[212] ≡ 212

```
{
 /* one per processor per node group. */
 typedef struct {
   _sint nodesN;         /* number of nodes currently in the node group */
   _sint localsizeN;     /* nr of Nodes allocated per PE */
   _sint newnodesN;
   short better2virt;    /* evidence points for "2 virt layers is better" */
   plural _network_D *boss;
 } _node_group_D;
}
```

This macro is invoked in definition 210.

One node descriptor is allocated on each processor for each virtualization layer of each node group in each network used. Only the `boss` pointer is quasi-singular — for the reason mentioned above. `lxN` and `lyN` describe the logarithmic size of the node block; they are quasi-singular while the network is in form 0.

Node Descriptor Type[213] ≡ 213

```
{
 /* one per node object. */
 typedef struct {
```

```
    _realness  exists;   /* Whether this Node is actually used or not */
    _bint      lxN;      /* number of PEs in the node block in x-direction */
    _bint      lyN;      /* dito, y-direction */
    plural _node_group_D *boss;
    _sint      meI;      /* my own node number */
  } _node_D;
  }
```

This macro is invoked in definition 210.

One interface descriptor is allocated on each processor for each connection interface in each virtualization layer of each node group in each network used. The `boss` pointer and the localsize of the connection array (`con_ls`) are quasi-singular. The `conN` field holds the number of connections that are attached to this interface. When this field is temporarily invalid, as after a `REPLICATE con INTO 0`, `CONNECT`, or `DISCONNECT`, it holds the `_sint` equivalent of -1. It is lazily updated when needed. The quasi-singular `work_per_con` field holds the work per connection done for the connections at this interface averaged over all nodes in the node group and accumulated over whole program run; it is valid only in replicate 0. The `wpc` contains the work per connection that is actually measured for each node in the current replication phase (during formA phases only).

214 *Interface Descriptor Type*[214] ≡

```
  {
  /* one per connection interface per node object */
  typedef struct {
    _sint con_ls;     /* local size of the connection array */
    _sint conN;       /* number of connections at this interface */
    plural _node_D *boss;
    _work work_per_con; /* accumulated work per connection average */
    _work wpc;        /* work per connection (this node and replication phase) */
  } _interface_D;
  #define invalid_conN ((_sint)(short)-1)
  }
```

This macro is invoked in definition 210.

One connection descriptor is allocated on each processor for each element of the local connection array described by each connection interface in each virtualization layer of each node group in each network used. This means one for each physically existing connection; a physically existing connection, though, may be logically non-existing because (a) it is in an unused machine segment or (b) it is in an unused part of a machine segment or (c) it is in an unused portion of a local connection array or (d) it has been deleted. The `boss` pointer is quasi-singular. The `meI` field is not always kept current; it is used only during the `REPLICATE` operation.

215 *Connection Descriptor Type*[215] ≡

```
  {
  /* one per (local) connection object */
  typedef struct {
    plural _interface_D *boss;
    _sint     meI;    /* only valid within REPLICATE procedure */
    _realness exists;
  } _connection_D;
  }
```

This macro is invoked in definition 210.

The following type is used as the object type of the remote part of connections of any type. Its structure is identical to that of a connection type without any data elements. This fact enables the compiler to handle remote connections exactly like objects of "yet another connection type" in many situations.

216 *Remote Connection Type*[216] ≡

```
{
typedef struct {
  _Gptr          _oe;  /* opposite end */
  _connection_D  _me_D;
} _remote_connection;
}
```
This macro is invoked in definition 210.

32.3 Auxiliary types and machine description

The following integer types `bool`, `sint`, and `bint` are just shorthands and are used almost everywhere. It is reasonable to always use the smallest possible integer type in the generated code, because operations on smaller types are more efficient on the PEs (although not on the ACU).

Auxiliary Types[217] ≡ 217
```
{
typedef unsigned char  _bool;
typedef unsigned short _sint;  /* "short int" */
typedef unsigned char  _bint;  /* "byte int" */
}
```
This macro is defined in definitions 217, 218, 219, 220, and 221.
This macro is invoked in definition 210.

`Gptr` (Mnemonic: Global pointer) is a type that can represent the address of any byte in the PE array memory. It cannot address the ACU.

Auxiliary Types[218] ≡ 218
```
{
typedef struct {
  plural char  *a;     /* local address on PE */
  _sint         pe;    /* PE number */
} _Gptr;  /* machine-global pointer (PE array only) */
}
```
This macro is defined in definitions 217, 218, 219, 220, and 221.
This macro is invoked in definition 210.

`realness` is the type that is used to indicate the status of network, node, or connection data objects: `nonexisting` (false) means "this is no network, node, or connection"; `existing` (true) means "this is a real network/node/connection"; `shadow` (true) means "this is in a block of an existing node or in a segment of an existing network replicate, but not the first PE", i.e., it is not the primary node but a shadow node. Often the distinction between `existing` and `shadow` is ignored and the `realness` is used simply as a boolean value.

Auxiliary Types[219] ≡ 219
```
{
typedef unsigned char _realness;
#define _existing    1
#define _shadow      2
#define _nonexisting 0
}
```
This macro is defined in definitions 217, 218, 219, 220, and 221.
This macro is invoked in definition 210.

Another auxiliary type, called **`work`**, is used to represent the work being performed at connections/nodes/etc. Since that work is measured in "ticks" (which are 80 nanoseconds each), we need an integer representation that can hold very large numbers.

Auxiliary Types[220] ≡ 220

```
{
typedef unsigned long long _work;  /* 64-bit integer */
}
```

This macro is defined in definitions 217, 218, 219, 220, and 221.
This macro is invoked in definition 210.

Finally, the auxiliary type `replication_type` is used to represent which combination of old and new form (formA, form0) the data has during a topology changing operation.

221 *Auxiliary Types*[221] ≡

```
{
typedef enum { _0_to_0, _0_to_A, _A_to_0, _A_to_A } _replication_type;
}
```

This macro is defined in definitions 217, 218, 219, 220, and 221.
This macro is invoked in definition 210.

We also need some constants that describe the size of the MasPar used according to our naming conventions. Here they are:

222 *Machine Descriptors*[222] ≡

```
{
#define debug
#ifdef debug
#define procN nproc
#define xprocN nxproc
#define yprocN nyproc
#define lprocN lnproc
#define lxprocN lnxproc
#define lyprocN lnyproc
#else
#define procN 16384
#define xprocN 128
#define yprocN 128
#define lprocN 14
#define lxprocN 7
#define lyprocN 7
#endif
}
```

This macro is invoked in definition 210.

33 Auxiliary MPL header files

This section contains a few more header files containing miscellaneous auxiliary material.

33.1 Forgotten MPL library routine prototypes

There are some procedures in the MPL library for which no prototype exists anywhere in the header files (MasPar Software Version 3.0). The file `mplforgotten.h` contains these missing definitions.

223 **lib/mplforgotten.h**[223] ≡

```
{/*
File: mplforgotten.h, prototypes for functions missing in MasPar header files
RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/
```

```
#ifndef MPLFORGOTTEN_H
#define MPLFORGOTTEN_H
void            ss_xfetch (int dy, int dx, plural void *src,
                           plural void *dest, int nbytes);
void            sp_xfetch (int dy, int dx, plural void *src,
                           plural void* plural dest, int nbytes);
void            ps_xfetch (int dy, int dx, plural void* plural src,
                           plural void *dest, int nbytes);
void            pp_xfetch (int dy, int dx, plural void* plural src,
                           plural void* plural dest, int nbytes);
void            ss_xsend  (int dy, int dx, plural void *src,
                           plural void *dest, int nbytes);
void            sp_xsend  (int dy, int dx, plural void *src,
                           plural void* plural dest, int nbytes);
void            ps_xsend  (int dy, int dx, plural void* plural src,
                           plural void *dest, int nbytes);
void            pp_xsend  (int dy, int dx, plural void* plural src,
                           plural void* plural dest, int nbytes);
#endif
}
```

This macro is attached to an output file.

33.2 Miscellaneous utility stuff

The file `libmisc.h` contains various small definitions that have no other place to go to.

The `assert` macro is to be used in the MPL code generated by the compiler for assertion testing. The `TRACE` macro can be used to generate tracing output; its behavior is controlled by the variable `tracelevel` (defined at the top of the main program): the higher the tracelevel, the more output is generated, tracelevel 0 means trace output is switched off. Each call of the macro must supply a parameter `l` between 0 and 5.

The `cat` macros implement token concatenation using the preprocessor. This is needed for the templates.

The macro `sgl` implements casts from plural to singular. The result is a random one of the values supplied by the plural argument, i.e., for a unique result the argument has to have the same value on all active processors.

lib/libmisc.h[224] ≡ 224

```
{/*
File: libmisc.h, miscellaneous stuff
RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/

#ifndef LIBMISC_H
#define LIBMISC_H

/* _assert(bool) macro: */
#ifndef NDEBUG
#define _assert(e) ((e)?1:fprintf(stderr,"\nOOPS !!  '%s', line %d\n",\
__FILE__,__LINE__))
#else
#define _assert(e)    1
#endif NDEBUG

#ifndef NOTRACE
```

```
extern int _tracelevel;
#define _TRACE(l,a) ((l<=_tracelevel) ? (printf a, fflush(stdout)) : 0)
#else
#define _TRACE(l,a)
#endif

#define _min(a,b) ((a)<(b)?(a):(b))
#define _max(a,b) ((a)>(b)?(a):(b))

#define _sgl(a) (proc[selectOne()].(a))

/* The way the ANSI C preprocessor works is absolutely weird...: */
#define _cat2(a,b) _cat2aux(a,b)
#define _cat3(a,b,c) _cat3aux(a,b,c)
#define _cat4(a,b,c,d) _cat4aux(a,b,c,d)
#define _cat2aux(a,b) a##b
#define _cat3aux(a,b,c) a##b##c
#define _cat4aux(a,b,c,d) a##b##c##d

#endif
}
```

This macro is attached to an output file.

34 The template method

34.1 Why ?

There are several cases in the CuPit code generation on the MasPar where over a considerably large amount of code there is only very little variation that is induced by the user program. Since there is *some* variation, the code pieces cannot be made part of the run time system but must be emitted by the actual code generation. Since they are large and have so little variation, on the other hand, it would be quite ugly to put them in ordinary LIDO and PTG rules. So we use the template method to generate these code pieces without cluttering the LIDO code.

34.2 What ?

The idea behind the template method is to generate the whole code piece in a way that resembles a procedure call: The type of code piece is given a name and everything that has to be varied within it is made a parameter. The C preprocessor is used to implement this pseudo code generation procedures.

34.3 How ?

For each type of code piece, we have an include file (with name suffix `.tpl`) that contains the text of the code piece. The parameters are implemented as preprocessor `#define`s so that parameter evaluation occurs automatically when the preprocessor processes the include file. The actual code generation must only define the parameters and include the include file; the preprocessor does the rest of the work when the generated MPL code is compiled. The files defined below have name suffix `.tplr` because they are written using C-Refine and have to be converted to `.tpl` files via the `crefine` preprocessor before they are included in a program generated by the compiler. This conversion is done by the makefiles defined on page 54.3ff.

34.4 Example

Think of using a template to generate the code for a simple integer or float addition. Of course this application is much too small for the template method to be sensible to implement it — it is for illustration only. The parameters in this case are the two operands (called `_a_` and `_b_`). The template file `add.tpl` might look like this

example add.tpl[225] ≡ 225
```
{
  _a_ + _b_
 #undef _a_
 #undef _b_
 }
```
This macro is NEVER invoked.

and in the code generation rule, for instance the following code might be generated

example add.tpl code[226] ≡ 226
```
{
 #define _a_   nrOfErrors
 #define _b_   1
 #include "add.tpl"
 }
```
This macro is NEVER invoked.

and the resulting code would be

example add.tpl result[227] ≡ 227
```
{
     nrOfErrors  +    1
 }
```
This macro is NEVER invoked.

35 INIT templates

Procedures to create the initial representation of data objects are needed for all types (one each). For networks and node groups (or node arrays) these are called only once at the beginning of the program run. For nodes, connections, and records they may be called during the rest of the program run, too. Most of these initialization procedures are generated directly by the code generation; the exceptions are intialization of a node group or node array and initialization of an ordinary array. For these two cases, templates are given here.

35.1 INIT node array

The procedure for node array initialization is generic in the type name of the node type. The same procedure is used for the initialization of node groups which are initially just arrays of size 0.

The procedure usually allocates as many PEs for each node as fit onto the machine without virtualization.

lib/NodeArrayInit.tplr[228] ≡ 228
```
{/* File: NodeArrayInit.tpl
    RCS: $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void _cat3(INIT_,_type_,_group) (plural _type_* plural* arr,
```

```
                           plural _node_group_D *descr,
                           plural _network_D* net_D, int arrsizeN)
{
  /* precondition:
        net_D is set up correctly and describes a network in form 0.
     postcondition:
        *arr is allocated with arrsizeN nodes, the nodes and all descriptors
        are initialized correctly.
  */
  int   blocksN;
  _bint lxblocksize, lyblocksize, lrowsN, lcolsN;
  _assert (!net_D->formA && arrsizeN >= 0);
  _TRACE (1, ("INITNodeGroup(%x, %d)\n", (int)arr, arrsizeN));
  descr->boss = net_D;
  descr->nodesN = arrsizeN;
  descr->newnodesN = 0;  /* the initial nodes are not 'new' nodes */
  if (arrsizeN == 0) {
    descr->localsizeN = 0;
    *arr = 0;
  }
  else {
    compute_block_size0 (net_D, descr, &lxblocksize, &lyblocksize);
    _assert (descr->localsizeN == 1);
    `allocate and initialize nodes;
  }

`allocate and initialize nodes:
   /* network is in form 0 */
   *arr = (plural _type_* plural)_getmem (sizeof(_type_), true);
   lcolsN = lxprocN - lxblocksize;
   lrowsN = lyprocN - lyblocksize;
   blocksN = _S(lcolsN + lrowsN);
   /* me_D.work is already 0 (initialized by _getmem) */
   `me_D.boss = descr;
   `me_D.lxN  = lxblocksize;
   `me_D.lyN  = lyblocksize;
   `me_D.meI = (ixproc >> lxblocksize) +
               (iyproc >> lyblocksize << lcolsN);
   /* the following is correct since in form0 always all PEs are used: */
   if (`me_D.meI >= arrsizeN)
     `me_D.exists = _nonexisting;
   else
     `me_D.exists = `is upper left block corner ? _existing : _shadow;
   if (`me_D.exists)
     _cat2(INIT_,_type_) (_sgl(*arr));

`me_D:
   (*arr)->_me_D

`is upper left block corner:
   (ixproc & _M(lxblocksize)) == 0 &&
   (iyproc & _M(lyblocksize)) == 0

}
#undef _type_
```

}

This macro is attached to an output file.

This initialization lays out the node blocks in a way that assigns node indices from left to right (increasing x coordinate), then top to bottom (increasing y coordinate), then increasing virtualization layer index. Unused node blocks may occur in the uppermost virtualization layer only.

35.2 INIT non-node arrays

The procedure for non-node array initialization is generic in the type name `_type_` of the array type, in the base type of the array `_basetype_`, and in the `_size_` of the array (maximum index plus one).

lib/ArrayInit.tplr[229] ≡ 229

```
{-
 /* File: ArrayInit.tpl
    RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void _cat2(INIT_,_type_) (plural _basetype_ * ME)
 {
   int i;
   _TRACE (2, ("INITArray(%x, %d)\n", (int)ME, _size));
   for (i = 0; i < _size_; i++)
     _cat2(INIT_,_basetype_) (ME+i);
 }
 #undef _type_
 #undef _basetype_
 #undef _size_
 }
```

This macro is attached to an output file.

36 REDUCTION templates

Procedures to implement the `REDUCTION` operation are needed for connections, nodes, and networks. A different procedure is needed for each of the categories. These procedures are generic in the type `_type_` of the value to be reduced but not in the type of connection etc., since the objects that contain the value to be reduced and the actual reduction operation can be described universally by a number of integer and pointer parameters.

36.1 REDUCTION connections

For `REDUCTION` operations on connections, a generic procedure needs the following static parameters: `nd_D` is a node descriptor that tells on which PEs there is an existing node and how large its node block is. `interf_D` is an interface descriptor that tells the number of connections to access locally in each PE (note that this "local" access involves communication if the connection is a remote connection). `base` is the local base address of the connection array. `field_offset` is the offset of the value to reduce within a connection object. `exists_offset` is the offset of the realness indicator within a connection object. `con_size` is the size of the connection type. `is_remote` indicates whether the actual connection objects must be accessed remotely or can be accessed locally. `cmb` is a pointer to the actual `REDUCTION` function. `result` is the address of the variable where the reduction result is to be stored; it must be plural, because a plurally indexed array may be used in the user program.

lib/ReductionCon.tplr[230] ≡ 230

{

```
/* File: ReductionCon.tpl
   RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/
plural void _cat3(p_REDUCTION_,_type_,_connections) (
   plural _node_D* nd_D, plural _interface_D* interf_D,
   register plural char* base, _sint field_offset, _sint exists_offset,
   register _sint con_size, _bool is_remote,
   plural _type_ (*cmb)(plural _type_*, plural _type_*),
   plural _type_* plural result)
{
  /* For each '_existing' node with at least one '_existing' connection,
     puts reduced value into *result on each shadow of that node.
     For nodes without connections, *result remains unchanged.
     active set: active existing and shadow nodes.
     Strategy: (1) First of all, reduce locally.
     (2) Then, since for every node block we can locally identify
     only the upper left corner, send boundary indicators
     to the right edge and lower left corner,
     (3) then reduce horizontally, (4) then from the
     upper left corner send boundary information to the lower left corner
     and mark all PEs in between as relevant for the next step,
     (5) then reduce vertically, (6) last, redistribute reduced value
     to all shadow nodes and assign it to *result.
  */
  plural _bool result_computed = false;
  register _sint  step, i;
  plural _realness node_exists = nd_D->exists;
  plural _realness connection_exists;
  plural _bool boundary_x = false, boundary_y = false;
  plural _bint lxN = nd_D->lxN;
  plural _bint lyN = nd_D->lyN;
  plural _bint xshadowI = ixproc & _M(lxN);
  plural _bint yshadowI = iyproc & _M(lyN);
  register plural _sint localsize = interf_D->con_ls;
  plural _type_ val, val2;
  _TRACE (4, ("ReductionCon (%x)\n", (int)base));
  if (is_remote)
    `do remote local reduction;
  else
    `do true local reduction;
  /* now result_computed shows whether local result is present in val */
  `set boundary indicators;
  `do reduction in x direction;
  if (xshadowI == 0)
    `do reduction in y direction;
  `redistribute reduced values;
  `place result;

`do remote local reduction:
   /* reduce into val,result_computed */
   for (i = 0; i < localsize; i++, base += con_size) {
     connection_exists = *(plural _realness*)(base+exists_offset);
     if (connection_exists) {
       ps_rfetch (`target con pe, `target con addr,
                  (plural void*)&val2, sizeof(_type_));
```

```
        if (result_computed)
          ‘reduce the local values;
        else
          ‘take the first local value;
      }
    }

‘target con pe:
    ((plural _remote_connection*)base)->_oe.pe

‘target_con_addr:
    ((plural _remote_connection*)base)->_oe.a + field_offset

‘do true local reduction:
    /* reduce into val,result_computed */
    for (i = 0; i < localsize; i++, base += con_size) {
      connection_exists = *(plural _realness*)(base+exists_offset);
      if (connection_exists) {
        val2 = *(plural _type_*)(base+field_offset);
        if (result_computed)
          ‘reduce the local values;
        else
          ‘take the first local value;
      }
    }

‘reduce the local values:
    /* call the reduction procedure: */
    val = (*cmb)(&val, &val2);

‘take the first local value:
    result_computed = true;
    val = val2;

‘set boundary indicators:
    /* mark rightmost column of each node block as boundary_x and mark
       PE yN-1 below existing node as boundary_y:
    */
    if (node_exists) {
      boundary_x = xshadowI == _M(lxN);
      boundary_y = yshadowI == _M(lyN);
    }

‘do reduction in x direction:
    step = 1;
    while (step < _S(lxN)) {
      if (boundary_x)
        xnetW[step].boundary_x = true;
      else
        ‘get and reduce x remote value;
      step <<= 1;
    }

‘get and reduce x remote value:
    ss_fetchx (step, (plural void*)&val, (plural void*)&val2, sizeof(_type_));
```

```
   connection_exists = xnetE[step].result_computed;
   if (connection_exists) {
      if (result_computed)
        ‘reduce the remote and local values;
      else
        ‘use the remote value;
   }

‘reduce the remote and local values:
   /* call the reduction procedure: */
   val = (*cmb)(&val, &val2);

‘use the remote value:
   result_computed = true;
   val = val2;

‘do reduction in y direction:
   /* active set: leftmost column of each node block */
   step = 1;
   while (step < _S(lyN)) {
     if (boundary_y)
       xnetN[step].boundary_y = true;
     else
       ‘get and reduce y remote value;
     step <<= 1;
   }

‘get and reduce y remote value:
   ss_fetchy (step, (plural void*)&val, (plural void*)&val2, sizeof(_type_));
   connection_exists = xnetS[step].result_computed;
   if (connection_exists) {
      if (result_computed)
        ‘reduce the remote and local values;
      else
        ‘use the remote value;
   }

‘redistribute reduced values:
   ss_xsendc (lyN, lxN, &val, sizeof (_type_),
              result_computed && node_exists == _existing);
   ss_xsendc (lyN, lxN, &result_computed, sizeof (_bool),
              result_computed && node_exists == _existing);

‘place result:
   /* set '*result' only where connections were actually present: */
   if (result_computed)
     *result = val;

}
#undef _type_
}
```

This macro is attached to an output file.

36.2 REDUCTION nodes

For `REDUCTION` operations on nodes, a generic procedure needs the following static parameters: `group_D` is a node group descriptor. `base` is the base address of the local node array. `field_offset` is the offset of the value to reduce within each node object. `descr_offset` is the offset of the node descriptor within each node object. `node_size` is the size of the node type. `cmb` is a pointer to the actual `REDUCTION` function. `result` is the address of the variable where the reduction result is to be stored.

The problem for this procedure is the different blocksize of each node in form A. This makes it impossible to directly compute the address of a partner node for a reduction step. Instead, all objects-to-reduce are sent to a vector that is distributed evenly on each segment and which is then reduced directly. The reduction is computed from the `_existing` nodes of each node block only. Each existing node computes the PE and address to send its value-to-be-reduced to from its own index and the segment size given in its boss' boss' descriptor. This method could be improved for reductions called while the network is in form 0 representation; they could use direct address computation instead of the indirection via the temporary vector.

lib/ReductionNode.tplr[231] ≡ 231

```
{
 /* File: ReductionNode.tpl
    RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void _cat3(p_REDUCTION_,_type_,_nodes) (plural _node_group_D* group_D,
    plural Interval slice,
    plural char* base, _sint field_offset, _sint descr_offset, _sint node_size,
    plural _type_ (*cmb)(plural _type_*, plural _type_*),
    plural _type_* plural result)
 {
   /* active set: all PEs belonging to active segments
   */
   plural _type_ *x, val;
   _sint          x_localsize;
   _sint          i, step;
   plural _sint   nodesN;
   plural _bool   result_computed = false;
   plural _bool   boundary_x = false, boundary_y = false;
   int            lxN = _sgl(`net_D.lxN),
                  lyN = _sgl(`net_D.lyN);
   plural _bint   x_in_seg_I = ixproc & _M(lxN),
                  y_in_seg_I = iyproc & _M(lyN);
   _TRACE (3, ("ReductionNode (%x)\n", (int)base));
   `adjust slice;
    nodesN = slice.max - slice.min + 1;
    if (nodesN == 0 || !`net_D.exists)
      return;  /* if slice is empty, there is nothing to reduce */
   `allocate x vector;
   `send values to x vectors;
   `reduce x vectors;
   `redistribute reduced values;
   `place result;

 `adjust slice:
    /* slice must contain only existing node indices: */
    if (slice.min < 0)
      slice.min = 0;
    if (slice.max >= group_D->nodesN)
```

```
     slice.max = (plural int)group_D->nodesN - 1;

 `allocate x vector:
    int xtarget, ytarget, itarget;
    if (`descr.exists == _existing)
      _lfold3 (_sgl(group_D->nodesN-1), lxN, lyN, xtarget, ytarget, itarget);
    x_localsize = itarget + 1;  /* smaller slices don't use all this room */
    x = (plural _type_*)p_alloca (sizeof(_type_) * x_localsize);
    _assert (x != 0);

 `send values to x vectors:
    plural _sint xtarget, ytarget, itarget,
                 meI;
    int localsize = proc[0].group_D->localsizeN;
    for (i = 0; i < localsize; i++, base += node_size) {
      if (`descr.exists == _existing &&
          `descr.meI >= slice.min && `descr.meI <= slice.max) {
          meI = `descr.meI - slice.min;
        _lfold3 (meI, lxN, lyN, xtarget, ytarget, itarget);
        sp_rsend (`target PE, base+field_offset, `target addr, sizeof(_type_));
      }
    }

 `descr:
    *(plural _node_D*)(base+descr_offset)

 `target PE:
    ((ixproc & ~_M(lxN)) + xtarget) +
    (((iyproc & ~_M(lyN)) + ytarget) << lxprocN)

 `net_D:
    *group_D->boss

 `target addr:
    (plural void* plural)(x + itarget)

 `reduce x vectors:
    `reduce locally;
    /* now result_computed shows whether local result is present in x[0] */
   `set boundary indicators;
   `do reduction in x direction;
   if (x_in_seg_I == 0)
     `do reduction in y direction;

 `reduce locally:
    i = 0;
    result_computed = `my slice node index < nodesN;
    for (i = 1; i < x_localsize; i++)
      if (`my slice node index < nodesN)
        x[0] = (*cmb)(x, x+i);

 `my slice node index:
    _unlfold3 (lxN, lyN, ixproc & _M(lxN), iyproc & _M(lyN), i)

 `set boundary indicators:
```

```
   /* mark rightmost column of each node block as boundary_x and mark
      PE yN-1 below existing node as boundary_y:
   */
   if (`net_D.exists) {
     boundary_x = x_in_seg_I == _M(lxN);
     boundary_y = y_in_seg_I == _M(lyN);
   }

`do reduction in x direction:
   step = 1;
   while (step < _S(lxN)) {
     if (boundary_x)
       xnetW[step].boundary_x = true;
     else
       `get and reduce x remote value;
     step <<= 1;
   }

`get and reduce x remote value:
   ss_fetchx (step, (plural void*)x, (plural void*)&val, sizeof(_type_));
   if (xnetE[step].result_computed) {
      if (result_computed)
        `reduce the remote and local values;
      else
        `use the remote value;
   }

`reduce the remote and local values:
   /* call the reduction procedure: */
   x[0] = (*cmb)(x, &val);

`use the remote value:
   result_computed = true;
   x[0] = val;

`do reduction in y direction:
   step = 1;
   while (step < _S(lyN)) {
     if (boundary_y)
       xnetN[step].boundary_y = true;
     else
       `get and reduce y remote value;
     step <<= 1;
   }

`get and reduce y remote value:
   ss_fetchy (step, (plural void*)x, (plural void*)&val, sizeof(_type_));
   if (xnetS[step].result_computed) {
      if (result_computed)
        `reduce the remote and local values;
      else
        `use the remote value;
   }

`redistribute reduced values:
```

```
    ss_xsendc (lyN, lxN, (plural void*)x, sizeof (_type_),
               result_computed && x_in_seg_I + y_in_seg_I == 0);
    ss_xsendc (lyN, lxN, (plural void*)&result_computed, sizeof (_bool),
               result_computed);

'place result:
    if (result_computed)
      *result = x[0];

}
#undef _type_
}
```

This macro is attached to an output file.

36.3 REDUCTION networks

For `REDUCTION` operations on networks, a generic procedure needs the following static parameters: `net_D` is the network descriptor. `field` is the address of the objects to be reduced. `cmb` is a pointer to the actual `REDUCTION` function. `result` is the address of the variable where the reduction result is to be stored.

This procedure is fairly simple, because no virtualization occurs and all segments have the same size. The reduction is computed from the `_existing` networks only.

232 **lib/ReductionNet.tplr**[232] ≡

```
{
/* File: ReductionNet.tpl
   RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/
plural void _cat3(p_REDUCTION_,_type_,_networks) (plural _network_D *net_D,
   Interval slice, plural _type_* field,
   plural _type_ (*cmb)(plural _type_*, plural _type_*),
   _type_* result)
{
  /* active set: all
  */
  plural _type_  val, val2;
  _sint          step;
  plural _bool   result_computed = false;
  _TRACE (2, ("ReductionNet (%x)\n", (int)field));
  'reduce networks;
  *result = proc[0].val;

'reduce networks:
   result_computed = net_D->exists == _existing &&
                     net_D->meI >= slice.min && net_D->meI <= slice.max;
   if (result_computed)
     val = *field;
   'do reduction in x direction;
   if (ixproc == 0)
     'do reduction in y direction;

'do reduction in x direction:
   step = _sgl(_S(net_D->lxN));
   while (step < xprocN) {
     if (ixproc + step < xprocN)
```

```
      ‘get and reduce x remote value;
    step <<= 1;
  }

‘get and reduce x remote value:
  if (xnetE[step].result_computed) {
     ss_fetchx (step, (plural void*)&val, (plural void*)&val2, sizeof(_type_));
     if (result_computed)
       ‘reduce the remote and local values;
     else
       ‘use the remote value;
  }

‘reduce the remote and local values:
  /* call the reduction procedure: */
  val = (*cmb)(&val, &val2);

‘use the remote value:
  result_computed = true;
  val = val2;

‘do reduction in y direction:
  step = _sgl(_S(net_D->lyN));
  while (step < yprocN) {
    if (iyproc + step < yprocN)
      ‘get and reduce y remote value;
    step <<= 1;
  }

‘get and reduce y remote value:
  if (xnetS[step].result_computed) {
     ss_fetchy (step, (plural void*)&val, (plural void*)&val2, sizeof(_type_));
     if (result_computed)
       ‘reduce the remote and local values;
     else
       ‘use the remote value;
  }

}
#undef _type_
}
```

This macro is attached to an output file.

37 WTA templates

The winner-takes-all procedures are very similar to the corresponding reduction procedures. Procedures to implement the `WTA` operation are needed for connections, nodes, and networks. A different procedure is needed for each of the categories. These procedures are generic in the type `_type_` of the value to be reduced but not in the type of connection etc., since the objects that contain the value to be reduced and the actual reduction operation can be described universally by a number of integer and pointer parameters. These procedures are more complicated than the reduction procedures in that not only a value must be propagated, but also the location it stems from.

37.1 WTA connections

For `WTA` operations on connections, a generic procedure needs the following static parameters: `nd_D` is a node descriptor that tells on which PEs there is an existing node and how large its node block is. `interf_D` is an interface descriptor that tells the number of connections to access locally in each PE (note that this "local" access involves communication if the connection is a remote connection). `base` is the local base address of the connection array. `field_offset` is the offset of the value to reduce within a connection object. `exists_offset` is the offset of the realness indicator within a connection object. `con_size` is the size of the connection type. `is_remote` indicates whether the actual connection objects must be accessed remotely or can be accessed locally. `wta` is a pointer to the actual `WTA` function. The function returns the address of the winning connection objects on all PEs containing a winning connection as a `plural void* plural` and zero on all others.

233 **lib/WtaCon.tplr**[233] ≡

```
{
 /* File: WtaCon.tpl
    RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void* plural _cat3(p_WTA_,_type_,_connections) (
    plural _node_D* plural nd_D,
    plural _interface_D* plural interf_D,
    plural char* base, _sint field_offset, _sint exists_offset,
    _sint con_size, _bool is_remote,
    plural _type_ (*wta)(plural _type_*, plural _type_*))
 {
   /* For each '_existing' node with at least one '_existing' connection,
      puts reduced value into *result on each shadow of that node.
      For nodes without connections, *result remains unchanged.
      active set: active existing and shadow nodes.
      Strategy: (1) First of all, reduce locally.
      (2) Then, since for every node block we can locally identify
      only the upper left corner, send boundary indicators
      to the right edge and lower left corner,
      (3) then reduce horizontally, (4) then from the
      upper left corner send boundary information to the lower left corner
      and mark all PEs in between as relevant for the next step,
      (5) then reduce vertically, (6) last, redistribute reduced value
      to all shadow nodes and assign it to *result.
   */
   typedef struct winner {
     _type_  val;
     _Gptr   where;
   };
   plural _bool result_computed = false;
   _sint  step, i;
   plural _realness node_exists = nd_D->exists;
   plural _realness connection_exists;
   plural _bool boundary_x = false, boundary_y = false, relevant = false;
   plural _bint lxN = nd_D->lxN;
   plural _bint lyN = nd_D->lyN;
   plural _bint xshadowI = ixproc & _M(lxN);
   plural _bint yshadowI = iyproc & _M(lyN);
   plural _sint localsize = interf_D->con_ls;
   plural struct winner val, val2;
   _TRACE (4, ("WtaCon (%x)\n", (int)base));
   if (is_remote)
```

```
    `do remote local reduction;
  else
    `do true local reduction;
  /* now result_computed shows whether local result is present in val */
  `set boundary indicators;
  `do reduction in x direction;
  if (xshadowI == 0)
    `do reduction in y direction;
  `redistribute winner descriptors;
  return (result_computed && val.where.pe == iproc ? val.where.a
                                                   : (plural void* plural)0);

`do remote local reduction:
   /* reduce into val,result_computed */
   val.where.pe  = iproc;
   val.where.a   = base;
   val2.where.pe = iproc;
   for (i = 0; i < localsize; i++, base += con_size) {
     connection_exists = *(plural _realness* plural)(base+exists_offset);
     if (connection_exists) {
       ps_rfetch (`target con pe, `target con addr,
                  (plural void*)&val2, sizeof(_type_));
       val2.where.a = base;
       if (result_computed)
         `reduce the local values;
       else
         `take the first local value;
     }
   }

`target con pe:
   ((plural _remote_connection* plural)base)->_oe.pe

`target_con_addr:
   ((plural _remote_connection* plural)base)->_oe.a + field_offset

`do true local reduction:
   /* reduce into val,result_computed */
   val.where.pe  = iproc;
   val.where.a   = base;
   val2.where.pe = iproc;
   for (i = 0; i < localsize; i++, base += con_size) {
     connection_exists = *(plural _realness* plural)(base+exists_offset);
     if (connection_exists) {
       val2.val = *(plural _type_* plural)(base+field_offset);
       val2.where.a = base;
       if (result_computed)
         `reduce the local values;
       else
         `take the first local value;
     }
   }

`reduce the local values:
   /* call the winner-takes-all procedure and store the new winner in 'val': */
```

```
    if (!(*wta)(&val.val, &val2.val))
      val = val2;

'take the first local value:
    result_computed = true;
    val = val2;

'set boundary indicators:
    /* mark rightmost column of each node block as boundary_x and mark
       PE yN-1 below existing node as boundary_y:
    */
    if (node_exists) {
      boundary_x = xshadowI == _M(lxN);
      boundary_y = yshadowI == _M(lyN);
    }

'do reduction in x direction:
    step = 1;
    while (step < _S(lxN)) {
      if (boundary_x)
        xnetW[step].boundary_x = true;
      else
        'get and reduce x remote value;
      step <<= 1;
    }

'get and reduce x remote value:
    ss_fetchx (step, (plural void*)&val, (plural void*)&val2,
               sizeof(struct winner));
    connection_exists = xnetE[step].result_computed;
    if (connection_exists) {
       if (result_computed)
         'reduce the remote and local values;
       else
         'use the remote value;
    }

'reduce the remote and local values:
    /* call the winner-takes-all procedure and store the new winner in 'val': */
    if (!(*wta)(&val.val, &val2.val))
      val = val2;

'use the remote value:
    result_computed = true;
    val = val2;

'do reduction in y direction:
    step = 1;
    while (step < _S(lyN)) {
      if (boundary_y)
        xnetN[step].boundary_y = true;
      else
        'get and reduce y remote value;
      step <<= 1;
    }
```

```
'get and reduce y remote value:
   ss_fetchy (step, (plural void*)&val, (plural void*)&val2,
              sizeof(struct winner));
   connection_exists = xnetS[step].result_computed;
   if (connection_exists) {
      if (result_computed)
        'reduce the remote and local values;
      else
        'use the remote value;
   }

'redistribute winner descriptors:
   ss_xsendc (lyN, lxN, &val.where, sizeof (_Gptr),
              result_computed && node_exists == _existing);
   ss_xsendc (lyN, lxN, &result_computed, sizeof (_bool),
              result_computed && node_exists == _existing);

}
#undef _type_
}
```

This macro is attached to an output file.

37.2 WTA nodes

For `WTA` operations on nodes, a generic procedure needs the following static parameters: `group_D` is a node group descriptor. `base` is the base address of the local node array. `field_offset` is the offset of the value to reduce within each node object. `descr_offset` is the offset of the node descriptor within each node object. `node_size` is the size of the node type. `wta` is a pointer to the actual `WTA` function. The function returns the address of the winning node objects on all PEs of the winning nodes' blocks and zero on all others.

The problem for this procedure is the different blocksize of each node in form A. This makes it impossible to directly compute the address of a partner node for a competition step. Instead, all objects-to-compare are sent to a vector that is distributed evenly on each segment and which is then reduced directly. The competition is computed from the `_existing` nodes of each node block only. Each existing node computes the PE and address to send its value-to-be-compared to from its own index and the segment size given in its boss' boss' descriptor. This method could be improved for WTAs called while the network is in form 0 representation; they could use direct address computation instead of the indirection via the temporary vector.

lib/WtaNode.tplr[234] ≡ 234

```
{
 /* File: WtaNode.tpl
    RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void* plural _cat3(p_WTA_,_type_,_nodes) (plural _node_group_D* group_D,
    plural Interval slice,
    plural char* base, _sint field_offset, _sint descr_offset, _sint node_size,
    plural Bool (*wta)(plural _type_*, plural _type_*))
 {
   /*
      active set: all PEs belonging to active segments
   */
   typedef struct winner {
```

```
    _type_  val;
    Gptr    where;
  };
  plural struct winner *x, val;
  _sint          x_localsize;
  _sint          i, step;
  plural _sint   nodesN;
  plural _bool   result_computed = false;
  plural _bool   boundary_x = false, boundary_y = false;
  int            lxN = proc[0].`net_D.lxN,
                 lyN = proc[0].`net_D.lyN;
  plural _bint   x_in_seg_I = ixproc & _M(lxN),
                 y_in_seg_I = iyproc & _M(lyN);
  _TRACE (3, ("WtaNode (%x)\n", (int)base));
  `adjust slice;
   nodesN = slice.max - slice.min + 1;
   if (nodesN == 0 || !`net_D.exists)
     return;  /* if slice is empty, there is nothing to reduce */
  `allocate x vector;
  `send values to x vectors;
  `reduce x vectors;
  `redistribute winner descriptors;
   return (`descr.nodesN > 0 &&
           iproc == x[0].where.pe ? x[0].where.a : (plural void* plural)0);

`adjust slice:
   /* slice must contain only existing node indices: */
   if (slice.min < 0)
     slice.min = 0;
   if (slice.max >= group_D->nodesN)
     slice.max = (plural int)group_D->nodesN - 1;

`allocate x vector:
   int xtarget, ytarget, itarget;
   if (`descr.exists == _existing)
     _lfold3 (_sgl(group_D->nodesN-1), lxN, lyN, xtarget, ytarget, itarget);
   x_localsize = itarget + 1;
   x = (plural _type_*)p_alloca (sizeof(struct winner) * x_localsize);
   _assert (x != 0);

`send values to x vectors:
   plural _sint xtarget, ytarget, itarget;
   val.where.pe = iproc;
   for (i = 0; i < group_D->localsizeN; i++, base += node_size) {
     if (`descr.exists == _existing &&
         `descr.meI >= slice.min && `descr.meI <= slice.max) {
       _lfold3 (`descr.meI, lxN, lyN, xtarget, ytarget, itarget);
       val.where.a = base+field_offset;
       val.val = *(plural _type_ *plural)(base+field_offset);
       sp_rsend (`target PE, (plural void*)&val, `target addr,
                 sizeof(struct winner));
     }
   }

`descr:
```

```
    *(plural _node_D*)(base+descr_offset)

 `target PE:
    ((ixproc & ~_M(lxN)) + xtarget) +
    ((iyproc & ~_M(lyN)) + ytarget) << lxprocN)

 `net_D:
    *group_D->boss

 `target addr:
    (plural void* plural)(x + itarget)

 `reduce x vectors:
    `reduce locally;
    /* now result_computed shows whether local result is present in x[0] */
   `set boundary indicators;
   `do reduction in x direction;
   if (x_in_seg_I == 0)
     `do reduction in y direction;

 `reduce locally:
    i = 0;
    result_computed = `my node index < group_D->nodesN;
    for (i = 1; i < x_localsize; i++)
      if (`my node index < group_D->nodesN  &&  !(*wta)(&x[0].val, &x[i].val))
        x[0] = x[i];

 `my node index:
    _unlfold3 (lxN, lyN, ixproc & _M(lxN), iyproc & _M(lyN), i)

 `set boundary indicators:
    /* mark rightmost column of each node block as boundary_x and mark
       PE yN-1 below existing node as boundary_y:
    */
    if (node_exists) {
      boundary_x = x_in_seg_I == _M(lxN);
      boundary_y = y_in_seg_I == _M(lyN);
    }

 `do reduction in x direction:
    step = 1;
    while (step < _S(lxN)) {
      if (boundary_x)
        xnetW[step].boundary_x = true;
      else
        `get and reduce x remote value;
      step <<= 1;
    }

 `get and reduce x remote value:
    ss_fetchx (step, (plural void*)x, (plural void*)&val, sizeof(struct winner));
    if (xnetE[step].result_computed) {
       if (result_computed)
         `reduce the remote and local values;
       else
```

```
      `use the remote value;
   }

`reduce the remote and local values:
   /* call the winner-takes-all procedure and store the new winner in x[0]: */
   if (!(*wta)(&x[0].val, &val.val))
     x[0] = val;

`use the remote value:
   result_computed = true;
   x[0] = val;

`do reduction in y direction:
   step = 1;
   while (step < _S(`net_D.lyN)) {
     if (boundary_y)
       xnetN[step].boundary_y = true;
     else
       `get and reduce y remote value;
     step <<= 1;
   }

`get and reduce y remote value:
   ss_fetchy (step, (plural void*)x, (plural void*)&val, sizeof(struct winner));
   if (xnetS[step].result_computed) {
      if (result_computed)
        `reduce the remote and local values;
      else
        `use the remote value;
   }

`redistribute winner descriptors:
   ss_xsendc (lyN, lxN, (plural void*)&x[0].where, sizeof (_Gptr),
              result_computed && `net_D.exists == _existing);
   /* superfluous!:
   ss_xsendc (lyN, lxN, (plural void*)&result_computed, sizeof (_bool),
              result_computed);   */

}
#undef _type_
}
```

This macro is attached to an output file.

37.3 WTA networks

For `WTA` operations on networks, a generic procedure needs the following static parameters: `net_D` is the network descriptor. `field` is the address of the objects that compete. `wta` is a pointer to the actual `WTA` function. The function returns true on all PEs of the winning segment and false on all others.

This procedure is fairly simple, because no virtualization occurs and all segments have the same size. The winner is computed from the `_existing` networks only.

235 **lib/WtaNet.tplr**[235] ≡

```
{
/* File: WtaNet.tpl
```

```
   RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/
plural _bool _cat3(p_WTA_,_type_,_networks) (plural _network_D net_D,
   Interval slice, plural _type_* field,
   plural Bool (*wta)(plural _type_* plural, plural _type_* plural))
{
  /* active set: all
  */
  plural _type_  val, val2;
  _sint          step;
  plural _sint   winnerI, winnerI2;
  plural _bool   result_computed = false;
  _TRACE (2, ("WtaNet (%x)\n", (int)field));
  `reduce networks;
  return (net_D.meI == winnerI);

`reduce networks:
   result_computed = net_D.exists == _existing &&
                     net_D.meI >= slice.min && net_D.meI <= slice.max;
   if (result_computed) {
     val = *field;
     winnerI = net_D.meI;
   }
   `do reduction in x direction;
   if (ixproc == 0)
     `do reduction in y direction;

`do reduction in x direction:
   step = _S(net_D.lxN);
   while (step < xprocN) {
     if (ixproc + step < xprocN)
       `get and reduce x remote value;
     step <<= 1;
   }

`get and reduce x remote value:
   if (xnetE[step].result_computed) {
      ss_fetchx (step, (plural void*)&val, (plural void*)&val2, sizeof(_type_));
      ss_fetchx (step, (plural void*)&winnerI, (plural void*)&winnerI2,
                 sizeof(winnerI));
      if (!result_computed || !(*wta)(&val, &val2))
        `use the remote as winner;
   }

`use the remote as winner:
   result_computed = true;
   val = val2;
   ss_fetchx (step, (plural void*)&winnerI, (plural void*)&winnerI,
              sizeof(winnerI));

`do reduction in y direction:
   step = _S(net_D.lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN)
       `get and reduce y remote value;
```

```
      step <<= 1;
    }

  `get and reduce y remote value:
    if (xnetS[step].result_computed) {
      ss_fetchy (step, (plural void*)&val, (plural void*)&val2, sizeof(_type_));
      ss_fetchy (step, (plural void*)&winnerI, (plural void*)&winnerI2,
                 sizeof(winnerI));
      if (!result_computed || !(*wta)(&val, &val2))
        `use the remote as winner;
    }

  }
  #undef _type_
  }
```

This macro is attached to an output file.

38 CONNECT template

The connect template is generic in the type `_type_` of connection made. The same version is used for arbitrary node types. The network is described via its network descriptor. The node groups are described by their base addresses, node sizes, slices used, and node descriptors. The nodes are described by a descriptor offset, interface offset, and interface descriptor offset.

When a reorganization of the network is necessary during the connect operation, neither the network as a whole, nor the node groups can possibly need reorganization. Only the interfaces affected by the connect statement may need to have their local connection arrays reallocated enlarged. Such reorganization is performed per node layer.

The procedure must always be called such that the remote part of the connections is placed at group 2. The procedure assumes that (1) all node blocks in each group are identically shaped, (2) memory allocation is singular for the node layers and the node's interfaces, (3) the interface array is and must be initialized to all non-existing and the localsize be set correctly even at non-existing nodes. No assumptions about holeless arrangement or linear numbering of nodes within groups are made.

236 **lib/Connect.tplr**[236] ≡

```
 {
  /* File: Connect.tpl
     RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
  */
  plural void _cat2(CONNECT_,_type_) (plural _network_D* net_D,
     plural char* base1, int node_size1, plural Interval slice1,
     plural _node_group_D* group_D1,
     int nd_D_offset1, int interf_D_offset1, int interf_offset1,
     plural char* base2, int node_size2, plural Interval slice2,
     plural _node_group_D* group_D2,
     int nd_D_offset2, int interf_D_offset2, int interf_offset2)
  {
    int slice1size,      /* how many relevant nodes in group 1 */
        slice2size;      /* dito, for group 2 */
    int cons_neededN,    /* total nr of cons to be made */
        cons_per_node1,  /* nr of cons to be made per node of group 1 */
        cons_per_node2,  /* dito, for group 2 */
        min_cons_freeN1, /* minimum nr of unused con objects per PE */
        min_cons_freeN2; /* dito, for node group 2 */
```

```
  int localsize1,      /* localsize of group 1 */
      localsize2;      /* dito, group 2 */
  int lxblocksize1,    /* log2 of width of node block in group 1 */
      lxblocksize2,    /* dito, for group 2 */
      lyblocksize1,    /* log2 of height of node block in group 1 */
      lyblocksize2,    /* dito, for group 2 */
      lblocksize1,     /* log2 of nr of PEs per node block */
      lblocksize2;     /* dito, for node group 2 */
  plural char* basebase1;
  plural char* basebase2; /* original value of base2 (base2 iterates) */
  int i, i_limit;         /* iteration variables */
  `check for form A;
  `adjust slices;
  _TRACE (2, ("CONNECT (%x %d..%d, %x %d..%d)\n",
              (int)base1, _sgl(slice1.min), _sgl(slice1.max),
              (int)base2, _sgl(slice2.min), _sgl(slice2.max)));
  `set variables;
  `find or make room for new connections;
  `make connections;

`check for form A:
   if (net_D->exists && net_D->formA) {
     fprintf (stderr, "CONNECT called while network is replicated\n");
     exit (14);
   }

`adjust slices:
   /* make slice1 and slice2 adhere to the following invariant:
      slice.min >= 0 && slice.max <= groupsize &&
      real_slice_size = slice.max - slice.min + 1
   */
   if (slice1.min < 0)                  slice1.min = 0;
   if (slice1.max >= group_D1->nodesN)  slice1.max =
                                            (plural int)group_D1->nodesN - 1;
   if (slice1.min > slice1.max)         slice1.max = slice1.min - 1;
   slice1size = (proc[0].slice1.max - proc[0].slice1.min + 1);
   if (slice2.min < 0)                  slice2.min = 0;
   if (slice2.max >= group_D2->nodesN)  slice2.max =
                                            (plural int)group_D2->nodesN - 1;
   if (slice2.min > slice2.max)         slice2.max = slice2.min - 1;
   slice2size = (proc[0].slice2.max - proc[0].slice2.min + 1);


`set variables:
   basebase1 = base1;
   if (`nd_D1.exists) {
     /* make casts plural -> singular from existing nodes only! */
     localsize1 = _sgl(group_D1->localsizeN);
     cons_per_node1 = slice2size;
     lxblocksize1 = _sgl(`nd_D1.lxN);
     lyblocksize1 = _sgl(`nd_D1.lyN);
     lblocksize1 = lxblocksize1 + lyblocksize1;
   }
   basebase2 = base2;
   if (`nd_D2.exists) {
```

```
      /* make casts plural -> singular from existing nodes only! */
      localsize2 = _sgl(group_D2->localsizeN);
      cons_per_node2 = slice1size;
      lxblocksize2 = _sgl(`nd_D2.lxN);
      lyblocksize2 = _sgl(`nd_D2.lyN);
      lblocksize2 = lxblocksize2 + lyblocksize2;
    }
    cons_neededN = cons_per_node1 * cons_per_node2;

`find or make room for new connections:
   /* the reorganization does not redistribute connections accross PEs,
      only reallocate them on the same PE.
      No connection changes its home processor.
   */
   for (i = 0; i < localsize1; i++, base1 += node_size1) {
     min_cons_freeN1 = `cons needed per PE 1;  /* as many as we may ever want */
     if (`nd_D1.exists && p_IntervalInOp (`nd_D1.meI, slice1))
       `determine minimum number of free connections1;
     if (min_cons_freeN1 < `cons needed per PE 1)
         `reorganize1;
   }
   base1 = basebase1;
   for (i = 0; i < localsize2; i++, base2 += node_size2) {
     min_cons_freeN2 = `cons needed per PE 2;  /* as many as we may ever want */
     if (`nd_D2.exists && p_IntervalInOp (`nd_D2.meI, slice2))
       `determine minimum number of free connections2;
     if (min_cons_freeN2 < `cons needed per PE 2)
         `reorganize2;
   }
   base2 = basebase2;

`determine minimum number of free connections1:
    plural _sint freeN = 0;
    plural _type_* cons;
    int k, k_limit = _sgl(`interf_D1.con_ls);
    cons = _sgl(`interf1);
    for (k = 0; k < k_limit; k++, cons++)
      if (!cons->_me_D.exists)
        freeN++;
    k = reduceMin16u (freeN);
    min_cons_freeN1 = _min (k, min_cons_freeN1);

`reorganize1:
    /* reallocate all connections at `interf1:
       (1) get new memory, (2) copy all connections, (3) store new pointers at
       remote ends, (4) release old memory, (5) initialize unused connections,
       (6) update interface descriptor, (7) update interface pointer
    */
    int new_con_ls = _sgl(`interf_D1.con_ls) +
                     (`cons needed per PE 1 - min_cons_freeN1);
    plural _type_ *new_interf = _getmem (new_con_ls * sizeof(_type_),
                                         true); /*(1),(5)*/
    plural _type_ *cons;           /* loop through connection array of a node */
    plural _type_* plural new_cons;  /* dito, for reorganization */
    int k, k_limit = _sgl(`interf_D1.con_ls);
```

```
   cons = _sgl(‘interf1);
   new_cons = new_interf;
   for (k = 0; k < k_limit; k++, cons++) {
     /* (2),(3): copy connection and store new remote pointer: */
     if (cons->_me_D.exists) {
       *new_cons = *cons;
       sp_rsend (cons->_oe.pe, (plural char*)&new_cons,
                 (plural char* plural)(cons->_oe.a +
                                       offsetof(_remote_connection,_oe.a)),
                 sizeof (plural _type_* plural));
       new_cons++;
     }
   }
   _freemem (_sgl(‘interf1)); /*(4)*/
   ‘interf_D1.con_ls = new_con_ls; /*(6)*/
   ‘interf_D1.conN = invalid_conN; /*(6)*/
   ‘interf1 = new_interf;          /*(7)*/

‘determine minimum number of free connections2:
   plural _sint freeN = 0;
   plural _remote_connection* rcons;
   int k, k_limit = _sgl(‘interf_D2.con_ls);
   rcons = _sgl(‘interf2);
   for (k = 0; k < k_limit; k++, rcons++)
     if (!rcons->_me_D.exists)
       freeN++;
   k = reduceMin16u (freeN);
   min_cons_freeN2 = _min (k, min_cons_freeN2);

‘reorganize2:
   /* reallocate all _remote_connections at ‘interf2:
      (1) get new memory, (2) copy all _remote_connections,
      (3) store new pointers at all data ends, (4) release old memory,
      (5) initialize new _remote_connections,
      (6) update interface descriptor, (7) update interface pointer
   */
   int new_con_ls = _sgl(‘interf_D2.con_ls) +
                    (‘cons needed per PE 2 - min_cons_freeN2);
   plural _remote_connection *new_interf =
            _getmem (new_con_ls * sizeof(_remote_connection), true); /*(1),(5)*/
   plural _remote_connection *rcons;
   plural _remote_connection* plural new_rcons;
   int k, k_limit = _sgl(‘interf_D2.con_ls);
   rcons = _sgl(‘interf2);
   new_rcons = new_interf;
   for (k = 0; k < k_limit; k++, rcons++) {
     /* (2),(3): copy _remote_connection and store new pointer at data: */
     if (rcons->_me_D.exists) {
       *new_rcons = *rcons;
       sp_rsend (rcons->_oe.pe, (plural char*)&new_rcons,
                 (plural char* plural)(rcons->_oe.a + offsetof(_type_,_oe.a)),
                 sizeof (plural _remote_connection* plural));
       new_rcons++;
     }
   }
```

```
   _freemem (_sgl(`interf2)); /*(4)*/
   `interf_D2.con_ls = new_con_ls; /*(6)*/
   `interf_D2.conN = invalid_conN; /*(6)*/
   `interf2 = new_interf;          /*(7)*/

`make connections:
   /* We reserve memory for the remote and the data connection addresses
      and store the addresses of the free connections there. The other end
      picks the address of its partner from this globally addressable
      memory. The address is computed as a cycle distribution over
      the connection number. The connection number is
        n = (nodeI1-slice1.min)*size(slice2) + (nodeI2-slice2.min)
      We always distribute the new connections evenly within each node block,
      no matter how uneven the overall connection distribution within
      this block may be. See 'find or make room' above.
   */
   int meet_localsize = cons_neededN/procN + ((cons_neededN & _M(lprocN)) != 0);
   plural _Gptr meet_cons[meet_localsize];
   plural _Gptr meet_rcons[meet_localsize];
   plural _Gptr meet_data;    /* object used for meetings */
   int nbpnI;                 /* normalized base partner_node index */
   plural int norm_p_nodeI;   /* normalized partner_node index */
   plural int global_conI;    /* global connection number */
   plural _sint meet_PE,   /* computed meeting point, PE number */
                meetI;     /* computed meeting point, local index */
   for (i = 0; i < localsize1; i++, base1 += node_size1)
     if (`nd_D1.exists && p_IntervalInOp (`nd_D1.meI, slice1))
       `write meet_cons;
   base1 = basebase1;
   for (i = 0; i < localsize2; i++, base2 += node_size2)
     if (`nd_D2.exists && p_IntervalInOp (`nd_D2.meI, slice2))
       `write meet_rcons and read meet_cons;
   base2 = basebase2;
   for (i = 0; i < localsize1; i++, base1 += node_size1)
     if (`nd_D1.exists && p_IntervalInOp (`nd_D1.meI, slice1))
       `read meet_rcons;
   base1 = basebase1;

`write meet_cons:
   /* active set: node1 blocks within slice1
      i is number of node1 virtualization layer, base1 points to these nodes.
   */
   plural _type_* plural cons = `interf1;
   plural j, j_limit = `cons needed per PE 1;
   nbpnI = 0;  /* within 0...(slice.max-slice.min) */
   for (j = 0; j < j_limit; j++, nbpnI += _S(lblocksize1)) {
     norm_p_nodeI = nbpnI + _unlfold2(lxblocksize1,
                                  ixproc & _M(lxblocksize1),
                                  iyproc & _M(lyblocksize1));
     global_conI = (`nd_D1.meI-slice1.min) * slice2size + norm_p_nodeI;
     if (global_conI < cons_neededN && norm_p_nodeI < slice2size) {
       while (cons->_me_D.exists)
         cons++;  /* find next free con */
       _fold2 (global_conI, procN, meet_localsize, meet_PE, meetI);
       meet_data.pe = iproc;
```

```
      meet_data.a = (plural char* plural)cons;
      sp_rsend (meet_PE, (plural char*)&meet_data,
                (plural char* plural)(meet_cons + meetI), sizeof (_Gptr));
      cons++;  /* so that we don't find the same free con again and again */
    }
  }
  _TRACE (3, ("  group 1: %x  con_array at %x, ls=%d\n", (int)base1,
             (int)_sgl(`interf1), _sgl(`interf_D1.con_ls)));

`write meet_rcons and read meet_cons:
  /* active set: node2 blocks within slice2
     i is number of node2 virtualization layer, base2 points to these nodes.
  */
  plural _remote_connection* plural rcons = `interf2;
  plural j, j_limit = `cons needed per PE 2;
  nbpnI = 0;  /* within 0...(slice.max-slice.min) */
  for (j = 0; j < j_limit; j++, nbpnI += _S(lblocksize2)) {
    norm_p_nodeI = nbpnI + _unlfold2(lxblocksize2,
                                  ixproc & _M(lxblocksize2),
                                  iyproc & _M(lyblocksize2));
    global_conI = norm_p_nodeI * slice2size + (`nd_D2.meI - slice2.min);
    if (global_conI < cons_neededN && norm_p_nodeI < slice1size) {
      while (rcons->_me_D.exists)
        rcons++;  /* find next free rcon */
      _fold2 (global_conI, procN, meet_localsize, meet_PE, meetI);
      meet_data.pe = iproc;
      meet_data.a = (plural char* plural)rcons;
      sp_rsend (meet_PE, (plural char*)&meet_data,
                (plural char* plural)(meet_rcons + meetI), sizeof (_Gptr));
      `make rcon;
    }
  }
  _TRACE (3, ("  group 2: %x  con_array at %x, ls=%d\n", (int)base2,
             (int)_sgl(`interf2), _sgl(`interf_D2.con_ls)));

`make rcon:
  ps_rfetch (meet_PE, (plural char* plural)(meet_cons + meetI),
            (plural char*)&meet_data, sizeof (_Gptr));
  rcons->_oe          = meet_data;
  rcons->_me_D.exists = _existing;
  rcons->_me_D.boss   = &`interf_D2;
  /* no need to set conI: memory is initialized with zeroes */

`read meet_rcons:
  /* active set: node1 blocks within slice1
     i is number of node1 virtualization layer, base1 points to these nodes.
  */
  plural _type_* plural cons = `interf1;
  plural j, j_limit = `cons needed per PE 1;
  nbpnI = 0;  /* within 0...(slice.max-slice.min+1) */
  for (j = 0; j < j_limit; j++, nbpnI += _S(lblocksize1)) {
    norm_p_nodeI = nbpnI + _unlfold2(lxblocksize1,
                                  ixproc & _M(lxblocksize1),
                                  iyproc & _M(lyblocksize1));
    global_conI = (`nd_D1.meI-slice1.min) * slice2size + norm_p_nodeI;
```

```
        if (global_conI < cons_neededN && norm_p_nodeI < slice2size) {
          while (cons->_me_D.exists)
            cons++;  /* find next free con */
          _fold2 (global_conI, procN, meet_localsize, meet_PE, meetI);
          `make con;
        }
      }

  `make con:
     ps_rfetch (meet_PE, (plural char* plural)(meet_rcons + meetI),
                (plural char*)&meet_data, sizeof (_Gptr));
     cons->_oe          = meet_data;
     cons->_me_D.exists = _existing;
     cons->_me_D.boss   = &`interf_D1;
     _cat2(INIT_,_type_) (cons);

  `cons needed per PE 1:
     (cons_per_node1 >> lblocksize1) + ((cons_per_node1 & _M(lblocksize1)) != 0)

  `cons needed per PE 2:
     (cons_per_node2 >> lblocksize2) + ((cons_per_node2 & _M(lblocksize2)) != 0)

  `interf_D1:
     *(plural _interface_D*)(base1+interf_D_offset1)

  `interf_D2:
     *(plural _interface_D*)(base2+interf_D_offset2)

  `interf1:
     *(plural _type_* plural*)(base1+interf_offset1)

  `interf2:
     *(plural _remote_connection* plural*)(base2+interf_offset2)

  `nd_D1:
     *(plural _node_D*)(base1+nd_D_offset1)

  `nd_D2:
     *(plural _node_D*)(base2+nd_D_offset2)

  }
  #undef _type_
  }
```

This macro is attached to an output file.

39 I/O templates

This section contains the templates that define the operations implementing the `<--` and `-->` operators. The only generic parameter needed is again the type `_type_` of the value to be transfered.

39.1 Input

The memory layout of the I/O area `x` is as follows: use one PE for each node to address (0..n), address node i of replicate 0 on PE number i, this is repeated for each replicate (0..k) by increasing the PE numbers consecutively. If the physical PEs do not suffice, begin again at proc[0].x[1]. The data at **x** must be laid out as if `slice.min...slice.max` was 0...n.

lib/Input.tplr[237] ≡ 237

```
{
 /* File: Input.tpl
    RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void _cat2(INPUT_,_type_) (
    plural _node_group_D* group_D, plural char* base, Interval slice,
    _sint field_offset, _sint descr_offset, _sint node_size, plural _type_ *x)
 {
   /* active set: all.
      The '_existing' nodes fetch the value from x and send it to
      the shadow nodes of their block.
   */
   plural _type_ val;
   plural int    vPE;  /* virtual-PE number */
   plural _realness ex;
   int localsizeN = _sgl(group_D->localsizeN),
       nodesN = _sgl(group_D->nodesN);
   int   i;
   `adjust slice;
   _TRACE (3, ("INPUT (%x %d..%d <-- %x)\n", (int)base, slice.min, slice.max,
               (int)x));
   for (i = 0; i < localsizeN; i++, base += node_size) {
     ex = `node_D.exists;
     if (p_IntervalInOp(`node_D.meI, slice)) {
       if (ex == _existing)
         `fetch value from x;
       `send value to shadow nodes;
       if (ex != _nonexisting)
         /* store val in existing or shadow node fields: */
         *(plural _type_*)(base+field_offset) = val;
     }
   }

 `adjust slice:
    /* make the slice adhere to the following invariant:
       slice.min >= 0 && slice.max <= groupsize &&
       real_slice_size = slice.max - slice.min + 1
    */
    if (slice.min < 0)          slice.min = 0;
    if (slice.max >= nodesN)    slice.max = nodesN - 1;
    if (slice.min > slice.max)  slice.max = slice.min - 1;
    nodesN = slice.max - slice.min + 1;

 `fetch value from x:
   vPE   = nodesN * `net_D.meI + (`node_D.meI - slice.min);
   ps_rfetch (vPE & _M(lprocN) /* = vPE % procN */,
              (plural void* plural)(x+(vPE>>lprocN)),
              (plural void*)&val, sizeof(_type_));

 `send value to shadow nodes:
   ss_xsendc (`node_D.lyN, `node_D.lxN, &val, sizeof (_type_),
              ex == _existing);
```

```
‘node_D:
   *(plural _node_D*)(base+descr_offset)

‘net_D:
   *group_D->boss

}
#undef _type_
}
```

This macro is attached to an output file.

39.2 Output

This procedure uses the I/O area in the same way as mentioned for the input procedure above.

238 **lib/Output.tplr**[238] ≡

```
{
/* File: Output.tpl
   RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/
plural void _cat2(OUTPUT_,_type_) (
   plural _node_group_D* group_D, plural char* base, Interval slice,
   _sint field_offset, _sint descr_offset, _sint node_size, plural _type_ *x)
{
  /* active set: all.
     The '_existing' nodes send the value to x.
  */
  plural _type_ val;
  plural int    vPE;
  int localsizeN = _sgl(group_D->localsizeN),
      nodesN = _sgl(group_D->nodesN);
  int i;
  ‘adjust slice;
  _TRACE (3, ("OUTPUT (%x %d..%d --> %x)\n", (int)base, slice.min, slice.max,
              (int)x));
  for (i = 0; i < localsizeN; i++, base += node_size)
    if (‘node_D.exists == _existing && p_IntervalInOp(‘node_D.meI, slice))
      ‘send value to x;

‘adjust slice:
   /* make the slice adhere to the following invariant:
      slice.min >= 0 && slice.max <= groupsize &&
      real_slice_size = slice.max - slice.min + 1
   */
   if (slice.min < 0)          slice.min = 0;
   if (slice.max >= nodesN)    slice.max = nodesN - 1;
   if (slice.min > slice.max)  slice.max = slice.min - 1;
   nodesN = slice.max - slice.min + 1;

‘send value to x:
  vPE   = nodesN * ‘net_D.meI + (‘node_D.meI - slice.min);
  val =  *(plural _type_*)(base+field_offset);
  sp_rsend (vPE & _M(lprocN) /* = vPE % procN */,
           (plural void*)&val, (plural void* plural)(x+(vPE>>lprocN)),
```

```
            sizeof(_type_));

'node_D:
   *(plural _node_D*)(base+descr_offset)

'net_D:
   *group_D->boss

}
#undef _type_
}
```

This macro is attached to an output file.

40 MERGE templates

These templates are instantiated for each connection type, node type, and network type used in the program. The resulting procedure, `a_MERGE_X`, is for calling the merging operations defined for the type `X` (and its components) — thus merging the network replicates — and for distributing the data from the first replicate to the others. The former is done, if the procedure is called with parameter `merge` equals true, the latter when it is called with parameter `distribute` equals true. Both can be chosen independently. An additional parameter `create_replicates` selects whether the distribution operation only re-distributes the data or *creates* replicates by distributing and modifying the descriptors as well. `create_replicates` has no meaning if `distribute` equals false. The only descriptor data that needs to be modified in the `create_replicates` case is the processor number of the `oe` pointers of all connections and remote connections. The change consists of adding the processor number of the upper left corner of the machine segment to the processor number already given in the `oe`.

Merging and (re-)distributing network, node, or connection data is relatively simple in the regular-machine-segmentation-network-replicate-data-layout (imagine smiley face here) the compiler uses, since the addresses of all pairs of corresponding objects are different only by multiples of constant differences in x and y processor numbers: For any object `x` in a network `net`, its correspondents in neighboring replicates of `net` can be found at the same local address `_S(net._me_D.lxN` processors to the left and to the right and `_S(net._me_D.lyN` processors up and down (given these neighboring replicates exist).

40.1 MERGE connections

The only generic parameter is the type `_type_` of the connection objects to be merged.

lib/MergeCon.tplr[239] ≡ 239

```
{
 /* File: MergeCon.tpl
    $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void _cat2(MERGE_,_type_) (plural _type_* ME,
    plural _type_ *YOU);  /* this prototype may be needed */

 plural void _cat2(a_MERGE_,_type_) (plural _network_D *net_D,
    plural _sint con_ls, plural _type_ *objs,
    _bool merge, _bool redistribute, _bool create_replicates)
 {
   /* active set: all
   */
   /* The whole connection array could be fetched at once. The resulting
      large communication packets may be more efficient although the
```

```
    descriptors have then to be moved also. But then we had to allocate
    additional memory (although only on the stack).
  */
  _sint step, i;
  _sint repN = _sgl(net_D->repN);
  int   lxN = _sgl(net_D->lxN),
        lyN = _sgl(net_D->lyN);
  _TRACE (4, ("MERGE_Con (%x, %d/%d/%d)\n", (int)objs, (int)merge,
             (int)redistribute, (int)create_replicates));
  for (i = 0; i < con_ls; i++, objs++)
    `merge and redistribute this connection;

`merge and redistribute this connection:
   if (merge) {
     plural _cat2(_type_,_0) obj2;
     plural _bool result_computed = net_D->exists;
     if (objs->_me_D.exists) {
       `do reduction in x direction;
       if (ixproc < _S(lxN))
         `do reduction in y direction;
     }
   }
   if (redistribute) {
     plural _bool result_computed = net_D->exists && net_D->meI == 0 &&
                                    objs->_me_D.exists;
     if (ixproc < _S(lxN))
       `do redistribution in y direction;
     `do redistribution in x direction;
     if (create_replicates)
       objs->_oe.pe += (ixproc & ~_M(lxN)) + ((iyproc & ~_M(lyN)) << lxprocN);
   }

`do reduction in x direction:
   step = _S(lxN);
   while (step < xprocN) {
     if (ixproc + step < xprocN && xnetE[step].result_computed)
       `get and reduce x remote value;
     step <<= 1;
   }

`get and reduce x remote value:
   ss_xfetch (0, step, (plural void*)objs, (plural void*)&obj2,
              sizeof(_cat2(_type_,_0)));
   _cat2(MERGE_,_type_) (objs, (plural _type_*)&obj2);

`do reduction in y direction:
   step = _S(lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN && xnetS[step].result_computed)
       `get and reduce y remote value;
     step <<= 1;
   }

`get and reduce y remote value:
   ss_xfetch (-step, 0, (plural void*)objs, (plural void*)&obj2,
```

```
              sizeof(_cat2(_type_,_0)));
   _cat2(MERGE_,_type_) (objs, (plural _type_*)&obj2);

 `do redistribution in y direction:
   step = _S(lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN && result_computed && `y_neighbor_I < repN)
       `put y remote value;
     step <<= 1;
   }

 `put y remote value:
   ss_xsend (-step, 0, (plural void*)objs, (plural void*)objs,
             create_replicates ? sizeof(_type_) : sizeof(_cat2(_type_,_0)));
   xnetS[step].result_computed = true;

 `do redistribution in x direction:
   step = _S(lxN);
   while (step < xprocN) {
     if (ixproc + step < xprocN && result_computed && `x_neighbor_I < repN)
       `put x remote value;
     step <<= 1;
   }

 `put x remote value:
   ss_xsend (0, step, (plural void*)objs, (plural void*)objs,
             create_replicates ? sizeof(_type_) : sizeof(_cat2(_type_,_0)));
   xnetE[step].result_computed = true;

 `x_neighbor_I:
   ((ixproc+step) >> lxN) + ((iyproc >> lyN) << (lxprocN-lxN))

 `y_neighbor_I:
   ((ixproc) >> lxN) + (((iyproc+step) >> lyN) << (lxprocN-lxN))

 }
 #undef _type_
 }
```

This macro is attached to an output file.

40.2 MERGE nodes

The node merge procedure template is significantly more tricky than the other templates above. It is generic not only in the type `_type_` of the network variable, but also in a macro called `_INTERFACES_`. When the template is to be instantiated, this macro must consist of a list of entries of the form `_I(t,o,i)`. One entry must be present for each connection interface `i` of the node type; `t` is the type of the respective interface and `o` is the type of the opposite end of the connections at this interface (i.e., one of `o` and `i` is always `_remote_connection` and the other is a user-declared connection type). This macro `_INTERFACES_` is used in the procedure wherever an iteration over the connection interfaces of the node is necessary. Before each use, an appropriate macro `_I` is defined so that each list entry will evaluate into a declaration, (partial) expression, procedure call, or whatever is needed. These macro expansions implement most of the functionality of the node merge procedure. An additional feature of this merge template is that it implements merging of the `wpc` values in the interface descriptors for the form A to form 0 transition in the rti code version; this merging is an averaging (summing and then dividing by the number of replicates).

240 **lib/MergeNode.tplr**[240] ≡

```
{
 /* File: MergeNode.tpl
    RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
 */
 plural void _cat2(MERGE_,_type_) (plural _type_ *ME,
    plural _type_ *YOU);  /* this prototype may be needed */

 plural void _cat2(a_MERGE_,_type_) (plural _network_D *net_D,
    plural _sint nd_ls, plural _type_ *objs,
    _bool merge, _bool redistribute, _bool create_replicates)
 {
   /* active set: all
   */
   _sint step, i;
   _bint lxN = _sgl(net_D->lxN),
         lyN = _sgl(net_D->lyN);
   _sint repN = _sgl(net_D->repN);
   _TRACE (3, ("MERGE_Node (%x, %d/%d/%d)\n", (int)objs, (int)merge,
               (int)redistribute, (int)create_replicates));
   for (i = 0; i < nd_ls; i++, objs++)
     `merge and redistribute this node and its connections;

 `merge and redistribute this node and its connections:
    if (merge) {
       plural _cat2(_type_,_0) obj2;
       plural _bool result_computed = net_D->exists;
      `do reduction in x direction;
      if (ixproc < _S(lxN))
        `do reduction in y direction;
      #define _I(_t,_o,_i) objs->_cat2(_i,_D).wpc /= repN;
      if (!redistribute) {
        _INTERFACES_
      }
      #undef _I
    }
    if (redistribute) {
      plural _bool result_computed = net_D->exists && net_D->meI == 0;
      if (ixproc < _S(lxN))
        `do redistribution in y direction;
      `do redistribution in x direction;
    }
    #define _I(_t,_o,_i) _cat2(a_MERGE_,_t)(net_D,objs->_cat2(_i,_D).con_ls,\
      _sgl(objs->_i), merge, redistribute, create_replicates);
    _INTERFACES_
    #undef _I

 `do reduction in x direction:
    step = _S(lxN);
    while (step < xprocN) {
      if (ixproc + step < xprocN && xnetE[step].result_computed)
        `get and reduce x remote value;
      step <<= 1;
    }
```

```
`get and reduce x remote value:
   ss_xfetch (0, step, (plural void*)objs, (plural void*)&obj2,
              sizeof(_cat2(_type_,_0)));
   _cat2(MERGE_,_type_) (objs, (plural _type_*)&obj2);
   #if _codetype_ == 1
   #define _I(_t,_o,_i) objs->_cat2(_i,_D).wpc +=\
                        xnetE[step].objs->_cat2(_i,_D).wpc;
   if (merge && !redistribute) {
     _INTERFACES_
   }
   #undef _I
   #endif

`do reduction in y direction:
   step = _S(lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN && xnetS[step].result_computed)
       `get and reduce y remote value;
     step <<= 1;
   }

`get and reduce y remote value:
   ss_xfetch (-step, 0, (plural void*)objs, (plural void*)&obj2,
              sizeof(_cat2(_type_,_0)));
   _cat2(MERGE_,_type_) (objs, (plural _type_*)&obj2);
   #if _codetype_ == 1
   #define _I(_t,_o,_i) objs->_cat2(_i,_D).wpc +=\
                        xnetS[step].objs->_cat2(_i,_D).wpc;
   if (merge && !redistribute) {
     _INTERFACES_
   }
   #undef _I
   #endif

`do redistribution in y direction:
   step = _S(lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN && result_computed && `y_neighbor_I < repN)
       `put y remote value;
     step <<= 1;
   }

`put y remote value:
   ss_xsend (-step, 0, (plural void*)objs, (plural void*)objs,
             create_replicates ? sizeof(_type_) : sizeof(_cat2(_type_,_0)));
   xnetS[step].result_computed = true;

`do redistribution in x direction:
   step = _S(lxN);
   while (step < xprocN) {
     if (ixproc + step < xprocN && result_computed && `x_neighbor_I < repN)
       `put x remote value;
     step <<= 1;
   }
```

```
‘put x remote value:
   ss_xsend (0, step, (plural void*)objs, (plural void*)objs,
             create_replicates ? sizeof(_type_) : sizeof(_cat2(_type_,_0)));
   xnetE[step].result_computed = true;

‘x_neighbor_I:
   ((ixproc+step) >> lxN) + ((iyproc >> lyN) << (lxprocN-lxN))

‘y_neighbor_I:
   ((ixproc) >> lxN) + (((iyproc+step) >> lyN) << (lxprocN-lxN))

}
#undef _type_
#undef _INTERFACES_
}
```

This macro is attached to an output file.

40.3 MERGE networks

The merge network procedure template uses a technique analogous to that used by the node merge template: It is generic not only in the type `_type_` of the network variable, but also in a macro called `_GROUPS_`. When the template is to be instantiated, this macro must consist of a list of entries of the form `_G(t,g)`. One entry must be present for each node group or node array `g` of the network type; `t` is the base type of the respective group or array type. This macro `_GROUPS_` is used in the procedure wherever an iteration over the node groups of the network is necessary.

241 **lib/MergeNet.tplr**[241] ≡

```
{
/* File: MergeNet.tpl
   RCS:  $Id: code1.fw,v 1.16 1994/11/07 10:51:26 prechelt Exp prechelt $
*/
plural void _cat2(MERGE_,_type_) (plural _type_ *ME,
   plural _type_ *YOU);  /* this prototype may be needed */

plural void _cat2(a_MERGE_,_type_) (plural _type_ *net,
   _bool merge, _bool redistribute, _bool create_replicates)
{
  /* active set: all
  */
  _sint step;
  _sint repN = _sgl(net->_me_D.repN);
  _bint lxN = _sgl(net->_me_D.lxN),
        lyN = _sgl(net->_me_D.lyN);
  _TRACE (3, ("MERGE_Net (%x, %d/%d/%d)\n", (int)net, (int)merge,
              (int)redistribute, (int)create_replicates));
  if (merge) {
    plural _cat2(_type_,_0) obj2;
    plural _bool result_computed = net->_me_D.exists;
    ‘do reduction in x direction;
    if (ixproc < _S(lxN))
      ‘do reduction in y direction;
  }
  if (redistribute) {
    plural _bool result_computed = net->_me_D.exists && net->_me_D.meI == 0;
    if (ixproc < _S(lxN))
```

```
      `do redistribution in y direction;
    `do redistribution in x direction;
    if (create_replicates)
      `modify meI;
  }
  #define _G(_t,_g) _cat2(a_MERGE_,_t)(&net->_me_D,\
    net->_cat2(_g,_D).localsizeN,_sgl(net->_g),merge,redistribute,\
    create_replicates);
  _GROUPS_
  #undef _G

`do reduction in x direction:
   step = _S(lxN);
   while (step < xprocN) {
     if (ixproc + step < xprocN && xnetE[step].result_computed)
       `get and reduce x remote value;
     step <<= 1;
   }

`get and reduce x remote value:
   ss_xfetch (0, step, (plural void*)net, (plural void*)&obj2,
              sizeof(_cat2(_type_,_0)));
   _cat2(MERGE_,_type_) (net, (plural _type_*)&obj2);

`do reduction in y direction:
   step = _S(lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN && xnetS[step].result_computed)
       `get and reduce y remote value;
     step <<= 1;
   }

`get and reduce y remote value:
   ss_xfetch (-step, 0, (plural void*)net, (plural void*)&obj2,
              sizeof(_cat2(_type_,_0)));
   _cat2(MERGE_,_type_) (net, (plural _type_*)&obj2);

`do redistribution in y direction:
   step = _S(lyN);
   while (step < yprocN) {
     if (iyproc + step < yprocN && result_computed && `y_neighbor_I < repN)
       `put y remote value;
     step <<= 1;
   }

`put y remote value:
   ss_xsend (-step, 0, (plural void*)net, (plural void*)net,
             create_replicates ? sizeof (_type_) : sizeof(_cat2(_type_,_0)));
   xnetS[step].result_computed = true;

`do redistribution in x direction:
   step = _S(lxN);
   while (step < xprocN) {
     if (ixproc + step < xprocN && result_computed && `x_neighbor_I < repN)
       `put x remote value;
```

```
      step <<= 1;
    }

  `put x remote value:
    ss_xsend (0, step, (plural void*)net, (plural void*)net,
              create_replicates ? sizeof (_type_) : sizeof(_cat2(_type_,_0)));
    xnetE[step].result_computed = true;

  `modify meI:
    /* although the meI and exists descriptors had been set correctly
       for all replicates in REPLICATE_, we have to recompute it now,
       because it was overwritten during `put remote value  above.
    */
    if (`meI < repN)
      net->_me_D.meI = `meI;
    else
      net->_me_D.exists = _nonexisting;

  `x_neighbor_I:
    ((ixproc+step) >> lxN) + ((iyproc >> lyN) << (lxprocN-lxN))

  `y_neighbor_I:
    ((ixproc) >> lxN) + (((iyproc+step) >> lyN) << (lxprocN-lxN))

  `meI:
    (ixproc >> lxN) + ((iyproc >> lyN) << (lxprocN-lxN))

  }
  #undef _type_
  #undef _GROUPS_
  }
```

This macro is attached to an output file.

41 EXTEND template

The `EXTEND` procedure template is generic in the type of the node it works for and the list of interfaces of that node type. There are four cases within this procedure: (1) shrinking a group is done by simply marking all nodes that shall be deleted and all their connections as non-existing, (2) extending a group of size 0 is the same as initialization, (3) extending a group that has enough free node blocks left is done by initializing these node blocks appropriately, and (4) extending a group that does not have enough free node blocks in its current representation requires a complete reorganization.

242 **lib/Extend.tplr**[242] ≡

```
  {
  plural void _cat2(EXTEND_,_type_) (plural _network_D *net_D,
      plural _type_* plural* group, plural _node_group_D* group_D,
      plural Int n)
  {
    _TRACE (1, ("EXTEND %x@%x BY %d\n", (int)group, (int)_sgl(*group), _sgl(n)));
    if (net_D->formA) {
      fprintf (stderr, "EXTEND not allowed for replicated networks\n");
      exit (15);
    }
    _assert (group_D->localsizeN <= 1); /* no node virtualization in form0 ! */
```

```
  _assert (n == _sgl(n));  /* n must in fact be singular */
  if (n < 0)
    `shrink group;
  else if (n > 0)
    `extend group;

`shrink group:
   if ((plural int)group_D->nodesN + n < 0) {
     fprintf (stderr, "EXTEND BY %d  called, but group has only %d nodes\n",
              _sgl(n), (plural int)_sgl(group_D->nodesN));
     exit (16);
   }
   group_D->nodesN += n;
   `mark nodes and all their connections as free;
   /* if no nodes left, free memory: */
   if ((plural int)group_D->nodesN == 0) {
     _freemem ((plural void*)_sgl(*group));
     *group = 0;
     group_D->localsizeN = 0;
   }
   group_D->newnodesN = 0;  /* deleting nodes makes new nodes old */

`mark nodes and all their connections as free:
   plural _bool must_vanish = `me_D.exists &&
                              `me_D.meI >= group_D->nodesN;
   if (must_vanish) {
     (*group)->_me_D.exists = _nonexisting;
     #define _I(_t,_o,_i) delete_connections (\
        (plural char*)_sgl(_cat2((*group)->,_i)), sizeof(_t),\
        _sgl(_cat3((*group)->,_i,_D).con_ls),\
        offsetof(_t,_oe), offsetof(_t,_me_D), offsetof(_o,_me_D));
     _INTERFACES_
     #undef _I
   }

`extend group:
   if (group_D->nodesN == 0)
     `init group;
   else {
     /* there is always a node at PE 0: */
     _bint lxblocksize = proc[0].(*group)->_me_D.lxN,
           lyblocksize = proc[0].(*group)->_me_D.lyN,
           lcolsN = lxprocN - lxblocksize,
           lrowsN = lyprocN - lyblocksize;
     int   blocksN = _S(lcolsN + lrowsN);
     group_D->newnodesN = n;
     if (`enough free node blocks)
       `add new nodes;
     else
       `reorganize and add new nodes;
   }

`init group:
   /* release old memory, if any: */
   if (*group != 0) {
```

```
    _assert (group_D->localsizeN != 0);
    _freemem ((plural void*)_sgl(*group));
    *group = 0;
    group_D->localsizeN = 0;
  }
  /* Create the node group: */
  _assert (n == _sgl (n));
  _cat3(INIT_,_type_,_group) (group, group_D, net_D, _sgl(n));

'enough free node blocks:
  blocksN >= group_D->nodesN + n

'add new nodes:
  plural _sint meI = (ixproc >> lxblocksize) +
                     (iyproc >> lyblocksize << lcolsN);
  plural _bool new = meI >= group_D->nodesN && meI < group_D->nodesN+n;
  group_D->nodesN += n;
  if (new) {
    'me_D.boss = group_D;
    'me_D.lxN  = lxblocksize;
    'me_D.lyN  = lyblocksize;
    'me_D.meI  = meI;
    'me_D.exists = 'is upper left block corner ? _existing : _shadow;
    _cat2(INIT_,_type_) (_sgl(*group));
  }

'is upper left block corner:
  (ixproc & _M(lxblocksize)) == 0 &&
  (iyproc & _M(lyblocksize)) == 0

'reorganize and add new nodes:
  plural _type_ *plural old_group = *group;
  plural _node_group_D old_group_D = *group_D;
  group_D->nodesN += n;
  _cat2(layout__,_type_) (_sgl(old_group_D.boss), _sgl(old_group_D.boss),
                          &old_group, &old_group_D, group, group_D, _0_to_0);
  _cat2(reconnect__,_type_) (_sgl(old_group_D.boss), _sgl(old_group_D.boss),
                          &old_group, &old_group_D, group, group_D);
  _cat2(release__,_type_) (_sgl(old_group_D.boss), _sgl(old_group_D.boss),
                          &old_group, &old_group_D, group, group_D);

'me_D:
  (*group)->_me_D

}
#undef _type_
#undef _INTERFACES_
}
```

This macro is attached to an output file.

42 REPLICATE templates

Templates for the `REPLICATE` operation are needed only for node and network types. Connection replication manipulates the descriptor of individual connection objects only and is thus implemented in the run time system.

42.1 REPLICATE node

The replicate node procedure template is generic in the type `_type_` of the node variable and in a macro `_INTERFACES_`. The latter is used in the same way as in the `MERGE` node template above. This procedure implements only the cases of node replication `INTO 0` and `INTO 1`, since replication into many is extremely complicated. The procedure works in the following steps: (1) check that the situation is legal (and exit with an error message if it is not), (2) mark the node itself as nonexisting, (3) delete all the connections of the node, (4) compute the new node numbers (`INDEX` values), (5) reorganize the group according to these new node numbers. For this last step, we would very much like to use the `layout_` and `reconnect_` procedure to do most of the work. This is, however, not directly possible, because they expect to get the address of the pointer to the node array (since they allocate a new node array), which is not available to the `REPLICATE_node` procedures. We use the following trick instead: We make a local copy of the node objects before we call `layout_` and tell it not to allocate a new node array (in fact, it is implemented to behave this way automatically when it finds the number of nodes to have decreased from the old group to the new.).

lib/ReplicateNode.tplr[243] ≡ 243

```
{
 plural void _cat2(REPLICATE_,_type_) (plural _type_ *ME, plural int into)
 {
   plural _realness ex;
   plural _bool delete_me;
   _sint new_nodesN;
   _TRACE (1, ("REPLICATE_Node %x INTO %d\n", (int)ME, _sgl(into)));
   `check legality;
   if (into == 1)
     return;
   /* now we are sure to have form0 and into=0. We must now include into the
      active set also those nodes that shall not be deleted:
   */
   all {
     delete_me = false;
     ex = ME->_me_D.exists;
   }
   delete_me = true;  /* on those PEs that were originally active */
   all {
     if (delete_me) {
       `delete node;
       `delete connections;
     }
     `compute new node numbers;
     `reorganize group;
   }

 `check legality:
    if (ME->_me_D.boss->boss->formA) {
      fprintf (stderr, "REPLICATE node not allowed in replicated networks\n");
      exit (17);
    }
    if (into < 0 || into > 1) {
      fprintf (stderr, "REPLICATE node INTO x only allowed for x=0 and x=1\n");
      exit (18);
    }
```

```
‘delete node:
   ME->_me_D.exists = _nonexisting;

‘delete connections:
   #define _I(_t,_o,_i) delete_connections (\
      (plural char*)_sgl(_cat2(ME->,_i)), sizeof(_t),\
      _sgl(_cat3(ME->,_i,_D).con_ls),\
      offsetof(_t,_oe), offsetof(_t,_me_D), offsetof(_o,_me_D));
   _INTERFACES_
   #undef _I

‘compute new node numbers:
   plural _sint newI;
   if (ME->_me_D.exists == _existing) {
     newI = enumerate();
     new_nodesN = reduceMax16u (newI) + 1;
   }
   if (ME->_me_D.exists) {
     ss_xsendc (ME->_me_D.lyN, ME->_me_D.lxN, &newI, sizeof (_sint),
                ME->_me_D.exists == _existing);
     ME->_me_D.meI = newI;
   }

‘reorganize group:
   plural _type_ old_nd = *ME;                 /* copy of the nodes */
   plural _type_ *plural group = ME;           /* plural pointer to nodes */
   plural _type_ *plural old_group = &old_nd;  /* plural pointer to copy */
   plural _node_group_D old_group_D;
   old_group_D = *ME->_me_D.boss;
   ME->_me_D.boss->nodesN = new_nodesN;
   ME->_me_D.boss->newnodesN = 0;  /* deleting nodes makes new nodes old */
   _cat2(layout__,_type_) (_sgl(old_group_D.boss), _sgl(old_group_D.boss),
        &old_group, &old_group_D, &group, _sgl(ME->_me_D.boss), _0_to_0);
   _cat2(reconnect__,_type_) (_sgl(old_group_D.boss), _sgl(old_group_D.boss),
        &old_group, &old_group_D, &group, _sgl(ME->_me_D.boss));

}
#undef _type_
#undef _INTERFACES_
}
```

This macro is attached to an output file.

42.2 REPLICATE network

The replicate network procedure template is generic in the type `_type_` of the network variable and in a macro `_GROUPS_`. The latter is used in the same way as in the `MERGE` network template above. Note that this procedure does not actually construct multiple replicates: It only constructs replicate 0 in the correct way; all other replicates are then created as relocated copies of replicate 0 by the `MERGE` procedure.

244 **lib/ReplicateNet.tplr**[244] ≡

```
{
plural void _cat2(REPLICATE_,_type_) (plural _type_* net,
        Interval repls, _bool to_formA)
{
```

```
  /* precondition:  *net represents valid network in either form 0 or form A.
     postcondition: *net represents valid network in new form with
        requested number of replicates. Individual exemplars are equivalent
        to input net or merged input net, respectively.
  */
  _replication_type replication_type;
  int i;
  plural _type_ old_net;
  _TRACE (1, ("REPLICATE_Net (%x --> %d..%d (to_formA = %d))\n", (int)net,
              repls.min, repls.max, (int)to_formA));
  `compute replication parameters;
  if (replication_type == _A_to_0)
    _cat2(a_MERGE_,_type_) (net, true, false, false);
  old_net = *net;  /* shallow copy, boss pointers corrected during layout__ */
  `ensure that always one net is formA and one form0;
  `decide number of replicates to use and make segments;
  /* foreach group: decide block sizes, layout, copy nodes and connections */
  #define _G(_t,_g) _cat2(layout__,_t) (&old_net._me_D,&net->_me_D,\
    &old_net._g,&old_net._cat2(_g,_D),&net->_g,&net->_cat2(_g,_D),\
    replication_type);
  _GROUPS_
  #undef _G
  /* foreach group: reconnect connections */
  #define _G(_t,_g) _cat2(reconnect__,_t) (&old_net._me_D,&net->_me_D,\
    &old_net._g,&old_net._cat2(_g,_D),&net->_g,&net->_cat2(_g,_D));
  _GROUPS_
  #undef _G
  /* foreach group: release old_net memory */
  #define _G(_t,_g) _cat2(release__,_t) (&old_net._me_D,&net->_me_D,\
    &old_net._g,&old_net._cat2(_g,_D),&net->_g,&net->_cat2(_g,_D));
  _GROUPS_
  #undef _G
  /* distribute data to replicates,
     modify net_D.meI and connection oe pointers: */
  if (net->_me_D.repN > 1)
    _cat2(a_MERGE_,_type_) (net, false, true, true);

`compute replication parameters:
  /* here `D still represents the old net */
  if (to_formA)
    `check to A replication;
  else
    `check to 0 replication;

`check to A replication:
   if (`D.formA) {
     replication_type = _A_to_A;
     fprintf (stderr, "REPLICATE net called for replicated network\n");
     exit (19);
   }
   else
     replication_type = _0_to_A;

`check to 0 replication:
   if (`D.formA)
```

```
      replication_type = _A_to_0;
    else
      replication_type = _0_to_0;  /* pure reorganization */

  `ensure that always one net is formA and one form0:
    if (replication_type == _0_to_0 || replication_type == _A_to_0) {
      old_net._me_D.formA = true;  /* a little cheating for _0_to_0 */
      net->_me_D.formA    = false;
    }
    else {
      _assert (replication_type == _0_to_A);
      net->_me_D.formA    = true;
    }
    if (repls.min <= 0 || repls.max <= 0) {
      fprintf (stderr,
               "number of network replicates must be positive\n");
      exit (10);
    }

  `decide number of replicates to use and make segments:
    /* this refinement could as well be a run time system procedure */
    if (replication_type == _A_to_0 || replication_type == _0_to_0) {
      /* there is exactly one segment, covering the whole machine: */
      `D.exists = _existing;
      `D.meI = 0;
      `D.repN = 1;
      `D.lrepN = 0;
      `D.lxN = lxprocN;
      `D.lyN = lyprocN;
    }
    else { /* make many segments */
      /* no sophisticated decision implemented yet.
         Should at least limit the replicates to fit into memory.
      */
      plural _bint me_xI, me_yI;
      _bint lxrepN;
      _bint lsegmentsize;
      _assert (replication_type == _0_to_0 || replication_type == _0_to_A);
      `D.lrepN = _log2(repls.max);  /* rounds UP! */
      `D.lrepN -= (_S(`D.lrepN) > repls.max);
      `D.repN = _S(`D.lrepN);
      if (`D.repN < repls.min) {  /* if rounded-down power of two is too small */
        `D.repN = repls.max;      /* use repls.max logically and */
        `D.lrepN++;               /* rounded-up power of two physically */
      }
      _assert (`D.repN >= repls.min && `D.repN <= repls.max);
      _assert (_S(`D.lrepN) >= `D.repN);
      lsegmentsize = lprocN - _sgl(`D.lrepN);
      `D.lyN = lsegmentsize >> 1;
      `D.lxN = lsegmentsize - `D.lyN;
      _assert (`D.lxN <= lxprocN && `D.lyN <= lyprocN);
      me_xI = ixproc >> `D.lxN;
      me_yI = iyproc >> `D.lyN;
      lxrepN = lxprocN - _sgl(`D.lxN);
      `D.meI = (me_yI << lxrepN) + me_xI;
```

```
      'D.exists = 'D.meI < 'D.repN ? _existing : _nonexisting;
    }

  'D:
    net->_me_D

  }
  #undef _GROUPS_
  #undef _type_
  }
```
This macro is attached to an output file.

The procedures called by the _G constructs embedded in the _GROUPS_ macro above have to be defined per node type. Therefore, a second template for the network replication is needed, to be instantiated for each node type. This template uses a technique similar to the one employed for _GROUPS_ above in order to handle iteration over connection interfaces; the generic parameter is called _INTERFACES_ (see also the replicate node and merge node templates above).

Three different procedures are needed, because there are three phases that must not overlap: The second part of the remote pointer handling can only be done if the first part has been done for the remote end of each connections. In the general case this can only be guaranteed by first performing the first part on all node groups before beginning with the second part (phase 1 and 2). Memory of the old node group and its connections must not be released before the remote pointers are completely established (phase 3).

lib/ReplicateNetNodes.tplr[245] ≡ 245

```
  {
   Layout Nodes[246]
   Reconnect Nodes[247]
   Release Node Memory[248]
  #undef _INTERFACES_
  #undef _type_
  }
```
This macro is attached to an output file.

The tasks to be solved by each layout_nodetype procedure are: (1) decide for each node in the group what its new node block size will be, (2) find a layout for the node blocks in segment 0 of the network, (3) set the group descriptor for the new node group, and (4) allocate memory and copy the nodes and their connections into their new node blocks. The remote pointers of the connections are handled as described in section 30.4.2. Most parts of the code are the same for form 0 and form A as target representation, although the form 0 case could be simplified.

The procedure can be used in three contexts:(1) during network replication, (2) during node group extension, and (3) during node replication (i.e., node deletion). In context (1), the old net is form A and the new is form 0 or vice versa and the number of nodes in the old and new group (nodesN) is the same. In context (2), both nets are form 0 and the old nodesN is always smaller than the new nodesN; the procedure has to create new nodes that not yet exist. In context (3), both nets are form 0 and the old nodesN is always larger than the new nodesN, but these additional old nodes do not actually exist anymore.

All three cases can be handled in mostly the same fashion, but differ in a few points. Most prominently, in context (3) there is no memory allocation for the new node array (see at REPLICATE nodes template). Another important difference is the handling of work_per_con and wpc measurement in context (1): work_per_con of the interfaces is used to compute the relative block sizes of the form A node blocks. In the rti version, wpc is used to update work_per_con and to count evidence points in the group descriptor's better2virt. All this is ignored in contexts (2) and (3).

Layout Nodes[246] ≡ 246

```
  {
```

```
plural void _cat2(layout__,_type_) (plural _network_D *old_net_D,
   plural _network_D *net_D,
   plural _type_* plural *old_nodes, plural _node_group_D *old_group_D,
   plural _type_* plural *nodes, plural _node_group_D *group_D,
   _replication_type replication_type)
{
  /* the following variables hold data for node i in proc[i]: */
  plural _sint size;    /* size of node block of new node i */
  plural _work work;    /* work of old node i's connections */
  plural _bint layer;   /* virtualization layer of new node i */
  plural _bint x0, y0;  /* upper left corner of node block of new node i */
  plural _bint lxN, lyN;/* log x and y size of node block of new node i */
  _bint layersN;        /* number of new virtualization layers */
  _bint lxblocksize, lyblocksize;  /* form 0 node block sizes */
  plural _bool is_replicate0 = old_net_D->exists && old_net_D->meI == 0;
  plural _bool willbe_replicate0 = net_D->exists && net_D->meI == 0;
  _sint new_nodesN = _sgl(group_D->nodesN), /* nodesN in new layout */
        newnodesN = _sgl(old_group_D->newnodesN), /* `young' nodes */
        old_nodesN = _sgl(old_group_D->nodesN), /* nodesN in old layout */
        existing_nodesN = old_nodesN > new_nodesN ? new_nodesN : old_nodesN;
  _sint layernodesN[2] = { old_nodesN-newnodesN, newnodesN };
  #define _I(_t,_o,_i) plural _sint _cat2(_i,_conN) = 0;\
    _sint _cat2(_i,_con_ls);\
    _work _cat2(_i,_wpc)[2] = {0, 0};
  _INTERFACES_
  #undef _I

 #define _normalizer 10000
  _assert (old_nodesN <= procN && new_nodesN <= procN);
  _assert (old_net_D->formA != net_D->formA ||
          old_nodesN != new_nodesN);
  _TRACE (2, ("layout__ (%x,%x)\n", (int)old_net_D, (int)_sgl(*old_nodes)));
  if (new_nodesN == 0) {
    if (old_nodesN == 0) {
      group_D->localsizeN = 0;
      *nodes = 0;
    }
    group_D->nodesN = 0;
    return;
  }
  /* most of the following logic is used for transformation into form A
     as well as transformation into form 0, although form 0 could be
     computed more efficiently. However, we decide to chose simpler
     code instead of faster execution here.
  */
  `prepare block information and correct boss pointers;
  if (replication_type == _A_to_0)
    `collect run time information;
  `compute block sizes;
  `compute block layout;
  `allocate and copy nodes;

`prepare block information and correct boss pointers:
   /* computes 'size', work, interface_D.conN, _i_conN, con_D.meI */
   int i;
```

```
   plural _type_ *old_nd = _sgl(*old_nodes);
   old_group_D->boss = old_net_D;  /* because old_net is a copy! */
   for (i = 0; i < _sgl(old_group_D->localsizeN); i++, old_nd++) {
       old_nd->_me_D.boss = old_group_D; /* because old_net is a copy! */
       if (old_nd->_me_D.exists)
         `handle this node layer;
   }
   old_nd = _sgl(*old_nodes);

`handle this node layer:
   if (is_replicate0) {
     `compute wpc sum and conN and conI;
     if (old_nd->_me_D.exists == _existing) {
       /* send conN values to _i_conN (which is initialized with 0): */
       #define _I(_t,_o,_i) router[old_nd->_me_D.meI]._cat2(_i,_conN) =\
           old_nd->_cat2(_i,_D).conN;
       _INTERFACES_
       #undef _I
     }
#if 0
     if (replication_type == _A_to_0) {
      printf ("in_wpc: %d/%d  out_wpc: %d/%d  work_per_con: %d,%d\n",
              (int)in_wpc[0]/_normalizer, (int)in_wpc[1]/_normalizer,
              (int)out_wpc[0]/_normalizer, (int)out_wpc[1]/_normalizer,
              (int)proc[0].old_nd->in_D.work_per_con/_normalizer,
              (int)proc[0].old_nd->out_D.work_per_con/_normalizer);
     }
     if (replication_type == _0_to_A) {
       printf ("conN:");
       all {if (iproc < old_nodesN)
              p_printf (" %2d/%2d", in_conN, out_conN);}
       printf ("\n");
     }
#endif
   }

`compute wpc sum and conN and conI:
   /* we must compute conN even when coming from formA, because it
      is too expensive to put the conN values into formA data upon creation
   */
   #define _I(_t,_o,_i) update_conI_conN((plural char*)_sgl(old_nd->_i),\
          sizeof(_t), offsetof(_t,_me_D), &old_nd->_cat2(_i,_D));
   _INTERFACES_
   #undef _I
   if (replication_type == _A_to_0) {
     #if _codetype_ == 1
       #define _I(_t,_o,_i) _cat2(_i,_wpc)[i] =\
               wpc_sum (&old_nd->_cat2(_i,_D)) / layernodesN[i];
       _INTERFACES_
       #undef _I
     #else
       ;
     #endif
   }
```

```
'collect run time information:
   /* executed for _A_to_0 only.
      1. collect better2virt evidence and adjust work_per_con.
         in codetype=1, formA there are always exactly two virt layers
         when new nodes exist and one virt layer otherwise
         We collect one evidence point in favor of separating new and
         old nodes for each time we find an interface whose wpc of new
         nodes differs by more than 20% from that of the old nodes;
         one point against if the difference is smaller.
      2. _i_wpc[0] := weighted_average (_i_wpc[0...1]);
         (to be written into new nodes' interface_Ds later)
   */
   #if _codetype_ == 1
     _work q;
     #define _I(_t,_o,_i) q = (100*_cat2(_i,_wpc)[0]) / (_cat2(_i,_wpc)[1]+1);\
         group_D->better2virt = old_group_D->better2virt + (q>120||q<80)?1:-1;\
         _cat2(_i,_wpc)[0] = (layernodesN[0]*_cat2(_i,_wpc)[0] + \
             layernodesN[1]*_cat2(_i,_wpc)[1])/(layernodesN[0]+layernodesN[1]);
     if (newnodesN > 0) {
       _INTERFACES_
     }
     #undef _I
   #else
     ;
   #endif

'compute block sizes:
   if (net_D->formA) {
     plural _work work = 0;
     plural _type_ *old_nd = _sgl(*old_nodes);
     'compute work for each node;
     size = compute_block_sizesA (work, new_nodesN, 'layer2 nodesN,
                                  procN>>_sgl(net_D->lrepN));
     if (_tracelevel >= 1) {
       printf ("blocksizeA:");
       if (iproc < old_nodesN)
         p_printf (" %d", (plural int)size);
       printf ("\n");
     }
     lxblocksize = proc[0].(old_nd->_me_D.lxN);
     lyblocksize = proc[0].(old_nd->_me_D.lyN);
   }
   else {
     compute_block_size0 (net_D, group_D, &lxblocksize, &lyblocksize);
     size = iproc < group_D->nodesN ? _S(lxblocksize+lyblocksize) : 0;
   }

'layer2 nodesN:
   !_sgl(net_D->formA) ||
   (_codetype_ == 2 && _sgl(old_group_D->better2virt) <= 0) ? 0 : newnodesN

'compute work for each node:
   /* executed for _0_to_A only */
   #if _noBalance_
     /* all nodes pretend to have the same work: */
```

```
    work = 100000;
  #elif _dumbBalance_
    /* work depends only on number of connections: */
    #define _I(_t,_o,_i) + _cat2(_i,_conN)
    work = 0 _INTERFACES_;
    #undef _I
  #else
    /* the REAL work computation: */
    #define _I(_t,_o,_i) + (proc[0].old_nd->_cat2(_i,_D).work_per_con *\
                           _cat2(_i,_conN))
    work = 0 _INTERFACES_;
    #undef _I
  #endif

'compute block layout:
  /* Precondition: 'size' reflects the block size of each node */
  compute_block_layout (size, new_nodesN, 'layer2 nodesN,
       _sgl(net_D->lxN), _sgl(net_D->lyN), _log2 (reduceMax16u(size)),
       &x0, &y0, &lxN, &lyN, &layer, &layersN);
  group_D->localsizeN = layersN;

'allocate and copy nodes:
  plural _type_ *nd, *old_nd;
  /* now either clear or allocate new nodes: */
  if (new_nodesN < old_nodesN)  /* called from 'REPLICATE node' context */
    p_memset (_sgl(*nodes), (plural int)0, sizeof(_type_));
  else
    /* _getmem() initializes .exists to false ! */
    *nodes = (plural _type_* plural)_getmem (sizeof(_type_) * layersN, true);
  old_nd = _sgl(*old_nodes);
  nd = _sgl(*nodes);
  'init _me_D of new _existing nodes;
  if (_sgl(!net_D->formA)) /* if new net is in form 0 */
    'send _existing old node to each new node and allocate connection arrays;
  else
    'fetch _existing old node to each new node and allocate connection arrays;
  'set work_per_con values;
  'send old connections to new connections;

'init _me_D of new _existing nodes:
  /* first init ALL nodes (existing, shadow, nonexisting) with
     neutral values, so that _sgl will never yield garbage.
  */
  plural _node_D nd_D;
  int i;
  nd_D.exists = _nonexisting;
  nd_D.boss   = group_D;
  for (i = 0; i < layersN; i++, nd++)
    nd->_me_D = nd_D;
  nd = _sgl(*nodes);
  if (iproc < group_D->nodesN) {
    nd_D.exists = _existing;
    nd_D.meI    = iproc;
    nd_D.lxN    = lxN;
    nd_D.lyN    = lyN;
```

```
    sp_rsend (x0 + (y0 << lxprocN), (plural char*)&nd_D,
              ((plural char* plural)&nd[layer]) + offsetof (_type_, _me_D),
              sizeof (_node_D));
  }

`send _existing old node to each new node and allocate connection arrays:
  plural _realness ex;
  int i;
  for (i = 0; i < _sgl(old_group_D->localsizeN); i++, old_nd++) {
    ex = old_nd->_me_D.exists;
    if (ex == _existing) {
      plural _sint xI, yI, layerI;
      plural _bint lxblocksN = lxprocN - lxblocksize,
                   lyblocksN = lyprocN - lyblocksize;
      _lfold3 (old_nd->_me_D.meI, lxblocksN, lyblocksN, xI, yI, layerI);
      if (is_replicate0)
        `send node;
    }
  }
  old_nd = _sgl(*old_nodes);
  for (i = 0; i < layersN; i++, nd++) {
    if (willbe_replicate0)
      `distribute _existing new node to rest of node block;
    `allocate connection arrays and set interfaces boss pointer;
  }
  nd = _sgl(*nodes);

`send node:
  plural int PE = (xI<<lxblocksize) + (yI<<(lyblocksize+lxprocN));
  sp_rsend (PE, (plural char*)old_nd, (plural char* plural)&nd[layerI],
            sizeof (_cat2(_type_,_0)));

`distribute _existing new node to rest of node block:
  ex = nd->_me_D.exists;
  _assert (nd->_me_D.exists != _shadow);
  if (ex == _existing)
    nd->_me_D.exists = _shadow;  /* change temporarily for copy */
  ss_xsendc (nd->_me_D.lyN, nd->_me_D.lxN, nd, sizeof (_type_),
             ex == _existing);  /* don't need interface data but _me_D !!! */
  if (ex == _existing)
    nd->_me_D.exists = _existing;

`allocate connection arrays and set interfaces boss pointer:
  #define _I(_t,_o,_i) if (iproc < new_nodesN && layer == i) \
    _cat2(_i,_con_ls) = reduceMax16u ((_cat2(_i,_conN)+_M(lxN+lyN)) >>\
      (lxN+lyN));\
    nd->_i = (plural _t* plural)_getmem(_cat2(_i,_con_ls)*sizeof(_t), true);\
    nd->_cat2(_i,_D).con_ls = _cat2(_i,_con_ls);\
    nd->_cat2(_i,_D).conN = invalid_conN;\
    nd->_cat2(_i,_D).boss = &nd->_me_D;
  _INTERFACES_
  #undef _I

`fetch _existing old node to each new node and allocate connection arrays:
  plural _realness ex;
```

```
      int i;
      for (i = 0; i < layersN; i++, nd++) {
        ex = nd->_me_D.exists;
        if (ex == _existing) {
          plural _sint xI, yI, layerI;
          plural _bint lxblocksN = lxprocN - lxblocksize,
                       lyblocksN = lyprocN - lyblocksize;
          _lfold3 (nd->_me_D.meI, lxblocksN, lyblocksN, xI, yI, layerI);
          if (willbe_replicate())
            `fetch or init the node;
        }
        if (willbe_replicate())
          `distribute _existing new node to rest of node block;
        `allocate connection arrays and set interfaces boss pointer;
      }
      nd = _sgl(*nodes);

   `fetch or init the node:
      plural int PE = (xI<<lxblocksize) + (yI<<(lyblocksize+lxprocN));
      if (nd->_me_D.meI < existing_nodesN)
        ps_rfetch (PE, (plural char* plural)&old_nd[layerI],
                   (plural char*)nd, sizeof (_cat2(_type_,_0)));
      else
        _cat2(INIT_,_type_) (nd);

   `set work_per_con values:
      int i;
      #define _I(_t,_o,_i) nd->_cat2(_i,_D).work_per_con =\
         _cat2(_i,_wpc)[0] + proc[0].old_nd->_cat2(_i,_D).work_per_con;
      for (i = 0; i < _sgl(group_D->localsizeN); i++, nd++) {
        _INTERFACES_
      }
      #undef _I
      nd = _sgl(*nodes);

   `send old connections to new connections:
      /* in order to be able to send the connections to their correct formA
         place, we use the con_D.meI at each connection and compute its new
         place using the global x0, y0, lxN, lyN, and layer.
      */
      int i;
      #define _I(_t,_o,_i) copy_connections((plural char*)old_nd,\
         (plural char*)nd, sizeof(_type_), offsetof(_type_,_me_D),\
         offsetof(_type_,_i), offsetof(_type_,_cat2(_i,_D)), sizeof(_t),\
         offsetof(_t,_me_D), offsetof(_t,_oe), x0, y0, lxN, lyN, layer, \
         is_replicate());
      for (i = 0; i < _sgl(old_group_D->localsizeN); i++, old_nd++) {
        _INTERFACES_
      }
      #undef _I
      old_nd = _sgl(*old_nodes);

   }
   }
```
This macro is invoked in definition 245.

The task to be solved by each `reconnect_`nodetype procedure is to execute the second phase of the remote connection pointer handling as described in section 30.4.2. Most of the parameters are not used in this procedure; they are present to maintain a common interface to all of the `layout_`, `reconnect_` and `release_` procedures.

The procedure can be used in three different contexts just like `layout_` above. See there for a description. In the `EXTEND` and `REPLICATE node` contexts, reconnecting is more difficult because intra-group connections have to be treated differently.

247 *Reconnect Nodes*[247] ≡

```
{
 plural void _cat2(reconnect__,_type_) (plural _network_D *old_net_D,
    plural _network_D *net_D,
    plural _type_* plural *old_nodes, plural _node_group_D *old_group_D,
    plural _type_* plural *nodes, plural _node_group_D *group_D)
 {
   int i;
   plural _type_ *nd = _sgl(*nodes);
   _TRACE (2, ("reconnect__ (%x,%x)\n", (int)old_net_D, (int)_sgl(*old_nodes)));
   if (net_D->meI != 0)
     return;  /* reconnect only on replicate 0 */
   if (old_group_D->nodesN == group_D->nodesN)
     `reconnect accross multiple groups;  /* for REPLICATE network */
   else
     `reconnect in one group; /* for EXTEND and REPLICATE node */

 `reconnect accross multiple groups:
   for (i = 0; i < _sgl(group_D->localsizeN); i++, nd++) {
     if (nd->_me_D.exists) {
       #define _I(_t,_o,_i) reconnect_connections ((plural char*)_sgl(nd->_i),\
         sizeof(_t), _sgl(nd->_cat2(_i,_D).con_ls),\
         offsetof(_t,_oe), offsetof (_t, _me_D), offsetof(_o,_oe));
       _INTERFACES_
       #undef _I
     }
   }

 `reconnect in one group:
    _assert ((net_D->formA | old_net_D->formA) == 0); /* both in form0 */
    _assert (group_D->localsizeN == 1); /* no virtualization in form0 */
    if (nd->_me_D.exists) {
      plural _type_ *old_nd = _sgl(*old_nodes);
      #define _I(_t,_o,_i) reconnect1_connections (\
         (plural char*)_sgl(old_nd->_i), (plural char*)_sgl(nd->_i),\
         sizeof(_t),\
         _sgl(old_nd->_cat2(_i,_D).con_ls),_sgl(nd->_cat2(_i,_D).con_ls),\
         offsetof(_t,_oe), offsetof (_t, _me_D), offsetof(_o,_oe));
      _INTERFACES_
      #undef _I
    }
 }
 }
```

This macro is invoked in definition 245.

The task to be solved by each `release_`nodetype procedure is to release the memory allocated for the old node group and its connections. Again, most parameters are not used by this procedure.

Release Node Memory[248] ≡ 248

```
{
 plural void _cat2(release__,_type_) (plural _network_D *old_net_D,
    plural _network_D *net_D,
    plural _type_* plural *old_nodes, plural _node_group_D *old_group_D,
    plural _type_* plural *nodes, plural _node_group_D *group_D)
 {
   int i;
   plural _type_ *old_nd = _sgl(*old_nodes);
   _TRACE (2, ("release__ (%x,%x)\n", (int)old_net_D, (int)_sgl(*old_nodes)));
   for (i = 0; i < old_group_D->localsizeN; i++, old_nd++) {
     #define _I(_t,_o,_i) _freemem (_sgl(old_nd->_i));
     _INTERFACES_
     #undef _I
   }
   _freemem (_sgl(*old_nodes));
   *old_nodes = 0;
 }
}
```

This macro is invoked in definition 245.

PART IV: Code Generation B — Now We Really Do It

This part describes the actual code generation. It is based on the strategies described above and uses the templates defined in part III and the run time system defined in part V.

43 General code generation definitions

43.1 Attributes and properties

249 *Code Generation Attributes*[249] ≡

```
{
ATTR code,      seqcode,      parcode,
     datacode,  moredatacode, descriptorcode,
     initcode,  printcode,    proccode, moreproccode, rticode,
     fetchcode, sendcode,
     smallcode, largecode,    seqslicecode, parslicecode,
     paramlist, arglist,
     Ptg      : PTGNode SYNT;
ATTR InhPtg   : PTGNode INH;
ATTR Interfaces,
     Groups   : PTGNode SYNT;
ATTR Size,
     Alignment : int SYNT;
ATTR InhAlignment
               : int INH;
CHAIN Offset   : int;
}
```

This macro is invoked in definition 334.

250 *Code Generation Properties*[250] ≡

```
{
Size,
Alignment,
Offset         : int;
}
```

This macro is invoked in definition 337.

The `code` attribute carries the unique total code product of symbols that have one. At many symbols however, several variants of code are needed (to be used in different contexts). In particular, most symbols have one form of code for a parallel context (attribute `parcode`) and a second one for a sequential context (attribute `seqcode`).

The code generation for data structures generates several code parts in parallel that are combined in multiple ways to generate to overall code: Declarations for "data elements" are generated in `datacode`, those for connection interface or node group element data are generated in `moredatacode`. The descriptors for the latter are generated in `descriptorcode`. Initialization code for the data elements is generated in the `initcode` attribute to be used for the generation of an initialization procedure for the type. The `printcode` attribute collects code generated for output procedures for the various types; these output procedures print the address, the data, and the descriptors of an object to standard output — procedures for the basic types are provided by the run time system. Note that calling these output procedures makes sense only with exactly one PE active.

The `proccode` attribute is used to collect the code for the several object subroutines in a type. `moreproccode` contains code generated "from nothing" that is needed to implement default cases. `rticode` contains code needed for the generation and collection of run-time information. `fetchcode` and `sendcode` contain the code for fetching and sending of the appropriate parts of a remote connection, respectively, and are used only in connection procedure generation.

The `smallcode` and `largecode` are used for reduction, winner-takes-all, and object procedures and for selection and subscription objects. For reduction and winner-takes-all functions, `smallcode` contains the actual reduction or WTA operator function while `largecode` contains the corresponding `a_REDUCTION` or `a_WTA` procedure(s) that actually carry out the respective operation over a set of values. For object procedures, `smallcode` contains the procedure itself and `largecode` contains the virtualized version (which calls the other). For selections, `smallcode` is the element selected and `largecode` is the `smallcode` of the object selected from. Both is to be used in the case of `ParVariableSelK` only. For subscriptions (with and without expression), `smallcode` is the object that is subscribed. This is to be used for `ParVariableK` objects only.

`seqslicecode` and `parslicecode` are used for `objects` to return the slice of a subscription on a named network object or a node group or node array object. This code is needed to produce the code of an object procedure call and the code for a `CONNECT` statement. The code always represents an object of type `Interval`.

`paramlist` and `arglist` are used to pass the formal parameter list and a corresponding actual parameter list (argument list) from a subroutine description to the `ObjProcedureDef` to be used for the generation of the virtualized `a_` version of the procedure.

`Ptg` and `InhPtg` are used to pass the PTG representations of identifiers (type identifiers in the latter case).

The attribute `Interfaces` stores a piece of program text that represents the list of connection interfaces of a node type. This list is represented as a number of entries of the form `_I(t,o,n)` separated by spaces. `n` is the name of a connection interface, `t` is its type and `o` is the type of the opposite end of the connections. `Groups` is a similar list representing the node groups and node arrays of a network type in the form `_G(bt,n)` where `bt` is the basetype(!) of the node group or node array.

The attribute and property `Size` stores for every basic, record, non-node array, and connection type `t` the value of `sizeof(t)` of the final implementation. The attribute and property `Alignment` stores for every basic, record, non-node array, and connection type its alignment requirements `a`. It means that objects of this type have to be stored at addresses divisible by `a`. The attribute is also used on the elements of record and connection types to compute the overall alignment. The chain `Offset` is used to compute the addresses of elements of record and connection types relative to the start of the object. The corresponding property stores this offset. `Size, Alignment`, and `Offset` are needed to compute optimal communication operations for fetching and sending the may-be-read and may-be-written sets of connection elements for remote connection operations. The alignment of a record is the maximum alignment of any of its elements. The actual alignment operation is computed as follows: The aligned address of an object with alignment `align` that has the proposed address `addr` is computed by `align(align,addr)` as follows (alignments are always powers of two!):
alignment computation[251] ≡ 251

```
{
 #define aligned(align,addr) ((addr&~(align-1))+((addr&(align-1))?align:0))
 }
```

This macro is invoked in definition 335.

43.2 Properties of predefined types

For the predefined types, the `Size` and `Alignment` properties must be set by the compiler before the code generation begins. This is done in a way similar to that used to define the predefined objects in section 16.3.

Set Properties of Predefined Types[252] ≡ 252

```
{
```

```
extern void SetPredefTypeSizeAlign ()
{
   SetSize      (BoolKey, 1, 1);
   SetAlignment (BoolKey, 1, 1);
   SetSize      (IntKey,  4, 4);
   SetAlignment (IntKey,  4, 4);
   SetSize      (Int1Key, 1, 1);
   SetAlignment (Int1Key, 1, 1);
   SetSize      (Int2Key, 2, 2);
   SetAlignment (Int2Key, 2, 2);
   SetSize      (RealKey, 4, 4);
   SetAlignment (RealKey, 4, 4);
   SetSize      (StringKey, 4, 4);
   SetAlignment (StringKey, 4, 4);

   SetSize      (IntervalKey,  8, 8);
   SetAlignment (IntervalKey,  4, 4);
   SetSize      (Interval1Key, 2, 2);
   SetAlignment (Interval1Key, 1, 1);
   SetSize      (Interval2Key, 4, 4);
   SetAlignment (Interval2Key, 2, 2);
   SetSize      (RealervalKey, 8, 8);
   SetAlignment (RealervalKey, 4, 4);
}
}
```

This macro is invoked in definition 339.

43.3 Auxiliary PTG definitions

253 **codehelp.ptg**[253] ≡

```
{
 Seq:    $ $
 Seq3:   $ $ $
 Seq4:   $ $ $ $
 Seq5:   $ $ $ $ $
 Seq6:   $ $ $ $ $ $
 List:   $ ", " $

 Comment:"  /* " $ " */"
 Str:    string []
 Int:    int []
}
```

This macro is attached to an output file.

43.4 General traversal order

Some of the CuPit code generation (exactly: PTG tree generation) is done in a text-order traversal of the syntax tree. We thus define a `CHAIN coded` that indicates how far we have gotten in that tree traversal.

254 *Coding Order*[254] ≡

```
{
 SYMBOL CupitProgram: allknown : VOID INH;
 SYMBOL CupitProgram: _allknown : VOID SYNT;
 CHAIN  coded: VOID;
```

```
SYMBOL CupitProgram COMPUTE
  CHAINSTART HEAD.coded = THIS.allknown;
  INH.allknown = THIS._allknown;
  SYNT._allknown =
    ORDER (SetPredefTypeSizeAlign (),
           Messag (NOTE, "coding starts")) DEPENDS_ON TAIL.known;
  Messag (NOTE, "everything is 'coded' now  ") DEPENDS_ON TAIL.coded;
END;
}
```

This macro is invoked in definition 334.

43.5 Overall program structure

Due to the defined-before-applied rule in the CuPit lanugage definition, the individual `CupitParts` can be implemented in the same order as they appear in the CuPit program. The `code` attribute is used in each of the parts to deliver the union of all code pieces generated from that part.

Cupit Program PTG[255] ≡ 255

```
{
 Program:
   "/* MPL implementation of CuPit program.\n"
   "   Generated by cupit.exe (code2.fw $Revision: 1.15 $)\n"
   "*/\n\n"
   "#include \"rts.h\"\n"
   "#define _dumbBalance_ " $5 "\n"
   "#define _noBalance_ " $6 "\n"
   "#define _wrongNodeVirt_ " $7 "\n"
   "#define _codetype_ " $4 "\n\n"
   "/************************** start of body *************************/\n\n"
   "visible int _tracelevel = 0;  /* tracing-output controller */\n"
   "visible int _randominit = 0;  /* random number generator init value */\n"
   "visible int _argsN;       /* number of float command line arguments */\n"
   "visible float _args[20];  /* the numeric command line arguments */\n"
   "visible int _namesN;      /* number of string command line arguments */\n"
   "visible int _nameoffsets[10]; /* string command line arg position */\n"
   "visible char _names[210]; /* the string command line arguments */\n"
   "visible void program ();  /* main procedure, visible for driver */\n"
   "visible void INIT ();     /* global INIT procedure, dito */\n"
   "void dump (String s);     /* debugging aid */\n\n"
   "visible void print_rusage ()\n"
   "{\n"
   "  struct mpRUsage_s ru;\n"
   "  mpGetRUsage (RUSAGE_SELF, &ru);\n"
   "  fprintf (stderr, \"dputime: %ds  imempgflts: %d  vcsw: %d  "
                      "ivcsw: %d\\n\",\n"
   "           ru.dpu.dr_dputime.tv_sec, ru.dpu.dr_imempgflt,\n"
   "           ru.dpu.dr_vcsw, ru.dpu.dr_vcsw);\n"
   "}\n"
   $1/*I/O procedures (this is wrong place if not for builtin type!)*/
   $2/*CupitParts*/
   $3/*INIT*/
   "\n/* end of compiler-generated MPL CuPit code */\n"
 GlobInit:
   "\nvoid INIT ()\n"
   "{\n" [IndentIncr]
```

```
  "   dpuTimerStart ();"
  "   _INITRANDOM (_randominit);"
  "   _initgetmem (0);"
  "   dpuTimerTicks2 ();" /* because something goes wrong(?) on first call */
  $   [IndentDecr]
  "\n}\n"
NetInitCall: [IndentNewLine] "INIT_" $1 " (&" $2 ");"
}
```

This macro is defined in definitions 255 and 258.
This macro is invoked in definition 336.

The code generated from a CuPit program consists of a fixed prologue, then the input and output assignment procedures for the types that need them, then the code generated from the program text itself, and last the global procedure that initializes all network variables,. The networks-initialization code is generated by a procedure `globINITgen` and the code for the input and output assignment procedures is generated by a procedure `IOprocgen`. Both will be defined below.

256 *Overall Program Generation*[256] ≡

```
{
RULE rCupitProgram :
  CupitProgram ::=  CupitParts
COMPUTE
  .initcode = PTGGlobInit (globINITgen (CONSTITUENTS InitDataId.NetVar
                           WITH (DefTblKeySet, DSunite, DSmk, DSempty)));
  .proccode =
     IOprocgen (CONSTITUENTS InputAssignment.Type
                WITH (DefTblKeySet, DSunite, DSmk, DSempty),
                CONSTITUENTS OutputAssignment.Type
                WITH (DefTblKeySet, DSunite, DSmk, DSempty));
  CupitProgram.code =
    PTGOut (PTGProgram (.proccode, CupitParts.code, .initcode,
               PTGInt (GetValue (codetype, 0)), PTGInt (dumbBalance),
               PTGInt (noBalance), PTGInt (wrongNodeVirt)));
END;

RULE rCupitParts0:
  CupitParts ::=
COMPUTE
  CupitParts.code = PTGNULL;
END;

RULE rCupitParts:
  CupitParts ::=  CupitParts CupitPart ';'
COMPUTE
  CupitParts[1].code =
    PTGSeq3 (CupitParts[2].code, PTGStr ("\n"),
             CupitPart CONSTITUENT (TypeDef.code, DataObjectDef.code,
                                    ProcedureDef.code, FunctionDef.code,
                                    ReductionFunctionDef.code,
                                    WtaFunctionDef.code)
                        SHIELD (TypeDef, DataObjectDef, ProcedureDef,
                                FunctionDef, ReductionFunctionDef,
                                WtaFunctionDef));
END;
}
```

This macro is invoked in definition 334.

To produce code for the input and output procedures, we take the set of types for which an input assignment (or output assignment, respectively) is found in the program and iterate it. The same technique is used to produce the initialization calls for all network variables in the program. The following module implements this functionality:

divgen.h[257] ≡ 257

```
{
 #include "deftblkeyset.h"
 #include "ptg_gen.h"

 PTGNode IOprocgen (DefTblKeySet in, DefTblKeySet out);
 PTGNode globINITgen (DefTblKeySet netvars);
 }
```

This macro is attached to an output file.

The code actually generated to produce an input or output procedure is just a template instantiation. Here is the PTG procedure `PTGIOProc` that describes how this template instantiation is created:

Cupit Program PTG[258] ≡ 258

```
{
 IOProc: "\n#define _type_ " $1
         "\n#include \"" $2 ".tpl\"\n"
 }
```

This macro is defined in definitions 255 and 258.
This macro is invoked in definition 336.

Given this, the procedure `IOprocgen` is a pretty simple iteration over the type lists:

divgen.c[259] ≡ 259

```
{
 #include "divgen.h"
 #include "pdl_gen.h"

 static PTGNode IOprocgen1 (DefTblKeySet s, PTGNode input_or_output)
 {
   PTGNode      result = PTGNULL;
   DefTableKey  key;
   int          sym;
   DSiterate (s);
   key = DSnext ();
   while (key != NoKey) {
     sym = GetSym (key, NoSym);
     _assert (sym != NoSym);
     result = PTGSeq (result,
                      PTGIOProc (PTGStr (SymString (sym)), input_or_output));
     key = DSnext ();
   }
   return (result);
 }


 PTGNode IOprocgen (DefTblKeySet in, DefTblKeySet out)
 {
   return (PTGSeq (IOprocgen1 (in, PTGStr ("Input")),
                  IOprocgen1 (out, PTGStr ("Output"))));
 }
```

```
PTGNode globINITgen (DefTblKeySet netvars)
{
  /* Attention: netvars contains NoKey as an element! */
  PTGNode      result = PTGNULL;
  DefTableKey  key;
  int          namesym, typesym;
  /* Generate the calls to the INITs for the individual network variables: */
  DSiterate (netvars);
  key = DSnext ();
  if (key == NoKey)
    key = DSnext (); /* skip NoKey element if this is not the end of the set */
  while (key != NoKey) {
    namesym = GetSym (key, NoSym);
    typesym = GetSym (GetType (key, NoKey), NoSym);
    _assert (namesym != NoSym);
    _assert (typesym != NoSym);
    result = PTGSeq (result,
                     PTGNetInitCall (PTGStr (SymString (typesym)),
                                     PTGStr (SymString (namesym))));
    key = DSnext ();
    if (key == NoKey)
      key = DSnext (); /* skip NoKey element if this is not the end of the set */
  }
  return (result);
}
}
```

This macro is attached to an output file.

44 Type definitions

From a type definition a whole lot of code sections may be generated: First of all the data type declaration itself (for any type), then the bare type without descriptors (for connection, node, and network types), then an initialization procedure (for record, connection, node, array, group, and network types), and the object procedures and object functions, if any, in a just-for-one-plural and in a for-all-local-plurals (i.e. virtualized) version. A constructor should also be generated, but this is not implemented.

260 *Type Definition PTG*[260] $\equiv$

```
{
 TypeDef:   "\ntypedef " $2 $1 ";\n"
 TypeDefHead:  [IndentNewLine] "/************************* " $1
                               " *************************/" [IndentNewLine]
```

Symbolic Type PTG[263]
Record Type PTG[265]
Node Type PTG[267]
Connection Type PTG[269]
Array Type PTG[271]
Group Type PTG[273]
Network Type PTG[275]
}

This macro is invoked in definition 336.

261 *Type Definition Generation*[261] $\equiv$

{

```
RULE rTypeDef :
  TypeDef ::=  'TYPE' NewTypeId 'IS' TypeDefBody 'END' OptTYPE
COMPUTE
  TypeDefBody.InhPtg = NewTypeId.Ptg;
  TypeDef.code = PTGSeq (PTGTypeDefHead (NewTypeId.Ptg),
                         Type Def Body Code[262]);
END;

Symbolic Type Generation[264]
Record Type Generation[266]
Node Type Generation[268]
Connection Type Generation[270]
Array Type Generation[272]
Group Type Generation[274]
Network Type Generation[276]
}
```

This macro is invoked in definition 334.

Type Def Body Code[262] ≡ 262

```
{TypeDefBody CONSTITUENT (SymbolicTypeDef.code, RecordTypeDef.code,
   NodeTypeDef.code, ConnectionTypeDef.code,
   ArrayTypeDef.code, GroupTypeDef.code, NetworkTypeDef.code)
}
```

This macro is invoked in definition 261.

44.1 Symbolic types

Symbolic types are coded straightforwardly as `enum` types.

Symbolic Type PTG[263] ≡ 263

```
{
SymbolicTypeDef:  "enum { " [IndentIncr] [IndentNewLine]
                  $ [IndentDecr] [IndentNewLine] "} "
}
```

This macro is invoked in definition 260.

Symbolic Type Generation[264] ≡ 264

```
{

RULE rSymbolicTypeDef :
  SymbolicTypeDef ::=  'SYMBOLIC' NewEnumIdList OptSEMICOLON
COMPUTE
  SymbolicTypeDef.code =
    PTGTypeDef (INCLUDING TypeDefBody.InhPtg,
                PTGSymbolicTypeDef (NewEnumIdList.code));
  SymbolicTypeDef.coded = ORDER (
    SetSize (.Type, 4, 0),
    SetAlignment (.Type, 4, 0));
END;

RULE rNewEnumIdList1 :
  NewEnumIdList ::=  NewEnumId
COMPUTE
  NewEnumIdList.code = NewEnumId.Ptg;
END;
```

```
RULE rNewEnumIdList :
  NewEnumIdList ::=  NewEnumIdList ',' NewEnumId
COMPUTE
  NewEnumIdList[1].code = PTGList (NewEnumIdList[2].code, NewEnumId.Ptg);
END;
}
```

This macro is invoked in definition 261.

44.2 Record types

Some of the rules given here for record types are also used by connection, node, and network types. Note that this implies that the rules must make special considerations for node groups, because these need additional descriptors (as opposed to ordinary data elements which do not).

265 *Record Type PTG*[265] ≡

```
{
RecordTypeDef:  "struct { /* RECORD */ " [IndentIncr]
                $ [IndentDecr] [IndentNewLine] "} "
RecordTypeInit: "\nplural void INIT_" $1 " (plural " $1 "* plural ME)"
                "\n{" [IndentIncr] $2 [IndentDecr] "\n}\n"
RecordTypePrint:
   "\nplural void p_pr" $1 " (String name, plural " $1 "* plural ME)"
   "\n{" [IndentIncr]
   [IndentNewLine] "printf (\"%s@%x=(\", name, (int)_sgl(ME));"
                   $2
   [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}\n"
Print:          [IndentNewLine] "p_pr" $1 "(\"" $2 "\", &ME->" $2 ");"
PrintGroup:     [IndentNewLine] "p_pr" $1 "(\"" $2 "\", &ME->" $2 ", "
                                "&ME->" $2 "_D);"
DataElemDef:    [IndentNewLine] $1 "  " $2 ";"
InitElem:       [IndentNewLine] "ME->" $1 " = " $2 ";"
InitRecordElem: [IndentNewLine] "INIT_" $2 " (&ME->" $1 ");"
InitNodeArrayElem: [IndentNewLine] "INIT_" $2 " (&ME->" $1 ", &ME->" $1 "_D, "
                                   "&ME->_me_D, " $3 ");"
NodeGroupDescr: [IndentNewLine] "_node_group_D  " $1 "_D;"
GroupAbbr:      "_G(" $1 "," $2 ") "
}
```

This macro is defined in definitions 265.
This macro is invoked in definition 260.

The code generated from a CuPit record type definition consists of four parts: the `datacode` is what will appear inside the type declaration itself, the `initcode` is the body of the initialization procedure for the type, the `printcode` is the kernel of the print procedure for the type, and the `proccode` consists of the complete code for the object subroutines of the record type.

266 *Record Type Generation*[266] ≡

```
{
RULE rRecordTypeDef :
  RecordTypeDef ::=  'RECORD' RecordElemDefList
COMPUTE
  .datacode =  CONSTITUENTS RecordDataElemDef.datacode
               WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .initcode =  CONSTITUENTS RecordDataElemDef.initcode
               WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .printcode = CONSTITUENTS RecordDataElemDef.printcode
```

```
              WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .proccode =  CONSTITUENTS (ObjProcedureDef.code, ObjFunctionDef.code)
              WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  RecordTypeDef.code =
    PTGSeq4 (PTGTypeDef (INCLUDING TypeDefBody.InhPtg,
                         PTGRecordTypeDef (.datacode)),
             PTGRecordTypeInit (INCLUDING TypeDefBody.InhPtg, .initcode),
             IF (ProducePrintcode,
               PTGRecordTypePrint (INCLUDING TypeDefBody.InhPtg, .printcode),
               PTGNULL),
             .proccode);
  RecordTypeDef.coded = ORDER (
    SetSize (.Type, RecordElemDefList.Offset, 0),
    SetAlignment (.Type, RecordElemDefList.Alignment, 1));
  CHAINSTART RecordElemDefList.Offset = 0;
END;

RULE rRecordElemDefList1 :
  RecordElemDefList ::=  RecordElemDef ';'
COMPUTE
  RecordElemDefList.Alignment = RecordElemDef.Alignment;
END;

RULE rRecordElemDefList :
  RecordElemDefList ::=  RecordElemDefList RecordElemDef ';'
COMPUTE
  RecordElemDefList[1].Alignment =
    IF (GT (RecordElemDefList[2].Alignment, RecordElemDef.Alignment),
      RecordElemDefList[2].Alignment, RecordElemDef.Alignment);
END;

SYMBOL RecordElemDef COMPUTE
  THIS.Alignment = 1;  /* for MERGE, FUNCTION, and PROCEDURE defs */
END;

RULE rRecordElemDef :
  RecordElemDef ::=  RecordDataElemDef
COMPUTE
  RecordElemDef.Alignment = RecordDataElemDef.Alignment;
END;


RULE rRecordDataElemDef :
  RecordDataElemDef ::=  TypeId InitElemIdList
COMPUTE
  RecordDataElemDef.datacode =
    PTGDataElemDef (TypeId.Ptg, InitElemIdList.datacode);
  RecordDataElemDef.initcode = InitElemIdList.initcode;
  RecordDataElemDef.printcode = InitElemIdList.printcode;
  RecordDataElemDef.Alignment =
    GetAlignment (TypeId.Key, 1) DEPENDS_ON RecordDataElemDef.coded;
  InitElemIdList.InhAlignment = RecordDataElemDef.Alignment;
END;

RULE rInitElemIdList :
```

```
  InitElemIdList ::=  InitElemIdList ',' InitElemId
COMPUTE
  TRANSFER InhAlignment;
  InitElemIdList[1].datacode =
    PTGList (InitElemIdList[2].datacode, InitElemId.datacode);
  InitElemIdList[1].descriptorcode =
    IF (EQ (InitElemId.descriptorcode, PTGNULL),
      PTGNULL,
      PTGSeq (InitElemIdList[2].descriptorcode, InitElemId.descriptorcode));
  InitElemIdList[1].initcode =
    PTGSeq (InitElemIdList[2].initcode, InitElemId.initcode);
  InitElemIdList[1].printcode =
    PTGSeq (InitElemIdList[2].printcode, InitElemId.printcode);
END;

RULE rInitElemIdList1 :
  InitElemIdList ::=  InitElemId
COMPUTE
  TRANSFER datacode, descriptorcode, initcode, printcode, InhAlignment;
END;

RULE rInitElemId1 :
  InitElemId ::=  NewElemId ':=' Expr
COMPUTE
  .Offset = aligned (InitElemId.InhAlignment, InitElemId.Offset);
  InitElemId.datacode = NewElemId.Ptg;
  InitElemId.descriptorcode = PTGNULL; /* initialized nodes not allowed */
  InitElemId.initcode = PTGSeq (
    IF (EQ (INCLUDING InitElemIdList.InhKind, RecordTypeK),
      PTGSeq (
        PTGInitRecordElem (NewElemId.Ptg, PTGKey (InitElemId.InhType)),
        PTGStr ("/* ERROR: Explicit Record initialization not implemented */")),
      PTGNULL),
    PTGInitElem (NewElemId.Ptg, Expr.parcode));
  InitElemId.printcode =
    IF (OR (EQ (InitElemId.InhKind, NodeArrayTypeK),
            EQ (InitElemId.InhKind, NodeGroupTypeK)),
      PTGPrintGroup (PTGKey (InitElemId.InhType), NewElemId.Ptg),
      PTGPrint (PTGKey (InitElemId.InhType), NewElemId.Ptg));
  InitElemId.Groups =
    IF (OR (EQ (InitElemId.InhKind, NodeArrayTypeK),
            EQ (InitElemId.InhKind, NodeGroupTypeK)),
      PTGGroupAbbr (PTGKey (InitElemId.InhType), NewElemId.Ptg),
      PTGNULL);
  NewElemId.coded = ORDER (
    Messag2 (NOTE, "Offset(%s)=%d", SymString (NewElemId.Sym), .Offset),
    SetOffset (NewElemId.Key, .Offset, 0));
  InitElemId.Offset =
    ADD (.Offset, GetSize (InitElemId.InhType, 0)) DEPENDS_ON InitElemId.coded;
END;

RULE rInitElemId0 :
  InitElemId ::=  NewElemId
COMPUTE
  .Offset = aligned (InitElemId.InhAlignment, InitElemId.Offset);
```

```
  InitElemId.datacode = NewElemId.Ptg;
  InitElemId.descriptorcode =
    IF (OR (EQ (InitElemId.InhKind, NodeArrayTypeK),
        EQ (InitElemId.InhKind, NodeGroupTypeK)),
      PTGNodeGroupDescr (NewElemId.Ptg),  /* see below */
      PTGNULL);
  InitElemId.initcode =
    IF (EQ (INCLUDING InitElemIdList.InhKind, RecordTypeK),
      PTGInitRecordElem (NewElemId.Ptg, PTGKey (InitElemId.InhType)),
    /* else */
    IF (EQ (INCLUDING InitElemIdList.InhKind, NodeArrayTypeK),
      PTGInitNodeArrayElem (
          NewElemId.Ptg, PTGKey (InitElemId.InhType),
          PTGInt (GetIval (GetVal (InitElemId.InhType, ErrorConst)))),
    /* else */
    IF (EQ (INCLUDING InitElemIdList.InhKind, NodeGroupTypeK),
      PTGInitNodeArrayElem (NewElemId.Ptg, PTGKey (InitElemId.InhType),
                            PTGInt (0)),
    /* else */
      PTGNULL)));
  InitElemId.printcode =
    IF (OR (EQ (InitElemId.InhKind, NodeArrayTypeK),
            EQ (InitElemId.InhKind, NodeGroupTypeK)),
      PTGPrintGroup (PTGKey (InitElemId.InhType), NewElemId.Ptg),
      PTGPrint (PTGKey (InitElemId.InhType), NewElemId.Ptg));
  InitElemId.Groups =
    IF (OR (EQ (InitElemId.InhKind, NodeArrayTypeK),
            EQ (InitElemId.InhKind, NodeGroupTypeK)),
      PTGGroupAbbr (PTGKey (GetType (InitElemId.InhType, NoKey)), NewElemId.Ptg),
      PTGNULL);
  NewElemId.coded = ORDER (
    Messag2 (NOTE, "Offset(%s)=%d", SymString (NewElemId.Sym), .Offset),
    SetOffset (NewElemId.Key, .Offset, 0));
  InitElemId.Offset =
    ADD (.Offset, GetSize (InitElemId.InhType, 0)) DEPENDS_ON InitElemId.coded;
END;
}
```

This macro is defined in definitions 266.
This macro is invoked in definition 261.

44.3 Node types

Node Type PTG[267] ≡ 267

```
{
NodeTypeDef: "struct { /* NODE */ " [IndentIncr]
             $1
             [IndentNewLine] "/* Interfaces: */" $2
             [IndentNewLine] "/* Descriptors: */" $3
             [IndentNewLine] "_node_D       _me_D;"
             [IndentDecr] [IndentNewLine] "} "
NodeTypeInit: "\nplural void INIT_" $1 " (plural " $1 "* ME)"
              "\n{" [IndentIncr] $2 [IndentDecr] "\n}\n"
NodeArrayTypeInit:
   "\n#define  _type_ " $1
   "\n#include \"NodeArrayInit.tpl\"\n"
```

```
NodeTypePrint:
   "\nplural void p_pr" $1 " (String name, plural " $1 "* plural ME)"
   "\n{" [IndentIncr]
   [IndentNewLine] "printf (\"\\n%s@%x=(\", name, (int)_sgl(ME));"
   [IndentNewLine] "p_pr_node_D (\"_me_D\", &ME->_me_D);"
   [IndentNewLine] "if (ME->_me_D.exists) {"
   [IndentIncr]        $2   [IndentDecr]
   [IndentNewLine] "}"
   [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}\n"
InterfaceDef:         [IndentNewLine] "plural " $1 "  " $2 ";"
remoteInterfaceDef:   [IndentNewLine] "plural _remote_connection  " $2 ";"
InterfaceDescr:       [IndentNewLine] "_interface_D  " $2 ";"
InitInterfaceElem:    [IndentNewLine] "ME->" $1 "_D.work_per_con = 1;"
                      [IndentNewLine] "ME->" $1 "_D.boss = &ME->_me_D;"
PrintInterfaceElem:
   [IndentNewLine] "p_pr" $1 "_interface (\"" $2 "\", ME->" $2 ", "
                   "&ME->" $2 "_D);"
InterfaceAbbr:        "_I(" $1 ",_remote_connection," $2 ") "
remoteInterfaceAbbr:  "_I(_remote_connection," $1 "," $2 ") "
MergeNodeTpl:         "\n#define _type_ " $1
                      "\n#define _INTERFACES_ " $2
                      "\n#include \"MergeNode.tpl\"\n"
ReplicateNetNodeTpl:  "\n#define _type_ " $1
                      "\n#define _INTERFACES_ " $2
                      "\n#include \"ReplicateNetNodes.tpl\"\n"
ExtendTpl:            "\n#define _type_ " $1
                      "\n#define _INTERFACES_ " $2
                      "\n#include \"Extend.tpl\"\n"
ReplicateNodeTpl:     "\n#define _type_ " $1
                      "\n#define _INTERFACES_ " $2
                      "\n#include \"ReplicateNode.tpl\"\n"
}
```

This macro is defined in definitions 267.
This macro is invoked in definition 260.

The code generated from a CuPit node type definition consists of six parts: The `datacode` (for data elements), `moredatacode`, (for connection interfaces) and `descriptorcode` (for connection interface descriptors) together form the variable parts of the body of the actual type declaration (the `datacode` alone is used for the `_0` version of the type); the `initcode` is the body of the initialization procedure for the type; the `printcode` is the kernel of the print procedure for the type; the `proccode` consists of the complete code for the object subroutines of the node type; finally, `moreproccode` contains a dummy `a_MERGE` procedure definition if there is no `MERGE` procedure defined for this type. The data declarations for the connection interfaces are separated in `moredatacode` in order to arrange the elements in a way that makes the `_0` type a prefix of the whole type.

268 *Node Type Generation*[268] ≡

```
{
 RULE rNodeTypeDef :
   NodeTypeDef ::=  'NODE' NodeElemDefList
 COMPUTE
   .datacode =         CONSTITUENTS NodeDataElemDef.datacode
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .moredatacode =     CONSTITUENTS NodeInterfaceElemDef.moredatacode
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .descriptorcode =   CONSTITUENTS NodeInterfaceElemDef.descriptorcode
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .initcode =         CONSTITUENTS (NodeDataElemDef.initcode,
```

```
                                   NodeInterfaceElemDef.initcode)
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .printcode =         CONSTITUENTS (NodeDataElemDef.printcode,
                                   NodeInterfaceElemDef.printcode)
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .proccode =          CONSTITUENTS (MergeProcDef.code, ObjProcedureDef.code,
                                   ObjFunctionDef.code)
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .moreproccode =
      IF (EQ (GetMergeDefs (INCLUDING TypeDefBody.InhKey, 0), 0),
        PTGMergeProcDef (INCLUDING TypeDefBody.InhPtg,
                         PTGStr ("\n  /* dummy */")),
        PTGNULL)  DEPENDS_ON NodeTypeDef.coded;
  .paramlist =         CONSTITUENTS (InterfaceIdList.Interfaces)
                       WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  NodeTypeDef.code =
    PTGSeq3 (
      PTGSeq ( /* typedefs: */
        PTGTypeDef (PTGSeq (INCLUDING TypeDefBody.InhPtg, PTGStr("_0")),
                    PTGRecordTypeDef (.datacode)),
        PTGTypeDef (INCLUDING TypeDefBody.InhPtg,
                    PTGNodeTypeDef (.datacode, .moredatacode,
                    .descriptorcode))),
      PTGSeq5 ( /* functions A: */
        PTGNodeTypeInit (INCLUDING TypeDefBody.InhPtg, .initcode),
        PTGNodeArrayTypeInit (INCLUDING TypeDefBody.InhPtg),
        IF (ProducePrintcode,
          PTGNodeTypePrint (INCLUDING TypeDefBody.InhPtg, .printcode),
          PTGNULL),
        PTGMergeNodeTpl (INCLUDING TypeDefBody.InhPtg, .paramlist),
        .moreproccode),
      PTGSeq4 ( /* functions B: */
        PTGReplicateNetNodeTpl (INCLUDING TypeDefBody.InhPtg, .paramlist),
        PTGExtendTpl (INCLUDING TypeDefBody.InhPtg, .paramlist),
        PTGReplicateNodeTpl (INCLUDING TypeDefBody.InhPtg, .paramlist),
        .proccode));
  /* dummy CHAINSTART (for InitElemId), chain is not actually needed: */
  CHAINSTART NodeElemDefList.Offset = 0;
END;

RULE rNodeDataElemDef :
  NodeDataElemDef ::=  TypeId InitElemIdList
COMPUTE
  NodeDataElemDef.datacode =
    PTGDataElemDef (TypeId.Ptg, InitElemIdList.datacode);
  /* InitElemIdList.moredatacode cannot occur */
  NodeDataElemDef.initcode = InitElemIdList.initcode;
  NodeDataElemDef.printcode = InitElemIdList.printcode;
  InitElemIdList.InhAlignment = 1; /* dummy to satisfy InitElemIdList */
END;

RULE rNodeInterfaceElemDef :
  NodeInterfaceElemDef ::=  InterfaceMode TypeId InterfaceIdList
COMPUTE
  .Dataloc = GetDataloc (TypeId.Key, NoMode)
```

```
            DEPENDS_ON INCLUDING CupitProgram.allknown;
  NodeInterfaceElemDef.moredatacode =
    IF (EQ (InterfaceIdList.InhMode, .Dataloc),
      PTGInterfaceDef (TypeId.Ptg, InterfaceIdList.moredatacode),
      PTGremoteInterfaceDef (TypeId.Ptg, InterfaceIdList.moredatacode));
  NodeInterfaceElemDef.descriptorcode =
      PTGInterfaceDescr (TypeId.Ptg, InterfaceIdList.descriptorcode);
  NodeInterfaceElemDef.initcode = InterfaceIdList.initcode;
  NodeInterfaceElemDef.printcode = InterfaceIdList.printcode;
END;

RULE rInterfaceModeIn :
  InterfaceMode ::=  'IN'
COMPUTE
END;

RULE rInterfaceModeOut :
  InterfaceMode ::=  'OUT'
COMPUTE
END;

RULE rInterfaceIdList1 :
  InterfaceIdList ::=  NewInterfaceId
COMPUTE
  .Dataloc = GetDataloc (InterfaceIdList.InhType, NoMode)
            DEPENDS_ON INCLUDING CupitProgram.allknown;
  InterfaceIdList.moredatacode =
    PTGSeq (PTGStr ("*"), NewInterfaceId.Ptg);
  InterfaceIdList.descriptorcode =
    PTGSeq (NewInterfaceId.Ptg, PTGStr ("_D"));
  InterfaceIdList.initcode = PTGInitInterfaceElem (NewInterfaceId.Ptg);
  InterfaceIdList.printcode =
    IF (EQ (InterfaceIdList.InhMode, .Dataloc),
      PTGPrintInterfaceElem (PTGKey (InterfaceIdList.InhType),
                             NewInterfaceId.Ptg),
      PTGPrintInterfaceElem (PTGStr ("_remote_connection"),
                             NewInterfaceId.Ptg));
  InterfaceIdList.Interfaces =
    IF (EQ (InterfaceIdList.InhMode, .Dataloc),
      PTGInterfaceAbbr (PTGKey (InterfaceIdList.InhType), NewInterfaceId.Ptg),
      PTGremoteInterfaceAbbr (PTGKey (InterfaceIdList.InhType),
                              NewInterfaceId.Ptg));
END;

RULE rInterfaceIdList :
  InterfaceIdList ::=  InterfaceIdList ',' NewInterfaceId
COMPUTE
  .Dataloc = GetDataloc (InterfaceIdList[2].InhType, NoMode)
            DEPENDS_ON INCLUDING CupitProgram.allknown;
  InterfaceIdList[1].moredatacode =
    PTGList (InterfaceIdList[2].moredatacode,
            PTGSeq (PTGStr ("*"), NewInterfaceId.Ptg));
  InterfaceIdList[1].descriptorcode =
    PTGList (InterfaceIdList[2].moredatacode,
            PTGSeq (NewInterfaceId.Ptg, PTGStr ("_D")));
```

```
  InterfaceIdList[1].initcode = /* Dummy !!! */
    PTGSeq (InterfaceIdList[2].initcode, PTGComment (NewInterfaceId.Ptg));
  InterfaceIdList[1].printcode =
    PTGSeq (InterfaceIdList[2].printcode,
    IF (EQ (InterfaceIdList[2].InhMode, .Dataloc),
      PTGPrintInterfaceElem (PTGKey (InterfaceIdList[2].InhType),
                             NewInterfaceId.Ptg),
      PTGPrintInterfaceElem (PTGStr ("_remote_connection"),
                             NewInterfaceId.Ptg)));
  InterfaceIdList[1].Interfaces =
    PTGSeq (InterfaceIdList[2].Interfaces,
      IF (EQ (InterfaceIdList[2].InhMode, .Dataloc),
        PTGInterfaceAbbr (PTGKey (InterfaceIdList[2].InhType),
                          NewInterfaceId.Ptg),
        PTGremoteInterfaceAbbr (PTGKey (InterfaceIdList[2].InhType),
                                NewInterfaceId.Ptg)));
END;
}
```

This macro is defined in definitions 268.
This macro is invoked in definition 261.

44.4 Connection types

Connection Type PTG[269] ≡ 269

```
{
 ConTypeDef:      "struct { /* CONNECTION */ " [IndentIncr]
                  $1
                  [IndentNewLine] "/* Descriptors: */" $2
                  [IndentDecr] [IndentNewLine] "} "
 ConTypeDescr:    [IndentNewLine] "_Gptr          _oe;"
                  [IndentNewLine] "_connection_D  _me_D;"
 ConnectProcDef:  "\n#define _type_ " $1
                  "\n#include \"Connect.tpl\"\n"
 MergeConTpl:     "\n#define _type_ " $1
                  "\n#include \"MergeCon.tpl\"\n"
 ConTypePrint:
    "\nplural void p_pr" $1 " (String name, plural " $1 "* plural ME)"
    "\n{" [IndentIncr]
    [IndentNewLine] "printf (\"\\n%s@%x=(\", name, (int)_sgl(ME));"
    [IndentNewLine] "p_pr_connection_D (\"_me_D\", &ME->_me_D);"
    [IndentNewLine] "if (ME->_me_D.exists) {"
    [IndentNewLine] "   p_pr_Gptr (\"_oe\", &ME->_oe);"
    [IndentIncr]        $2   [IndentDecr]
    [IndentNewLine] "}"
    [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}"
    "\n"
    "\nplural void p_pr" $1 "_interface (String name, plural " $1 "* plural ME,"
    [IndentNewLine] "      plural _interface_D* plural descr)"
    "\n{" [IndentIncr]
    [IndentNewLine] "char* n = \"0\";"
    [IndentNewLine] "int  i;"
    [IndentNewLine] "printf (\"\\n%s@%x=(\", name, (int)_sgl(ME));"
    [IndentNewLine] "p_pr_interface_D (\"_D\", descr);"
    [IndentNewLine] "for (i = 0; i < descr->con_ls; i++, ME++) {"
    [IndentNewLine] "  *n = (i % 64) + '0';"
```

```
    [IndentNewLine] "  p_pr" $1 " (n, ME);"
    [IndentNewLine] "}"
    [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}\n"
}
```

This macro is defined in definitions 269.
This macro is invoked in definition 260.

The code generated from a connection type definition consists of five parts: the `datacode` is what will appear inside the type declaration itself, the constant `descriptorcode` is what distinguishes the full type from the `_0` type, the `initcode` is the body of the initialization procedure for the type, the `printcode` is the kernel of the print procedure for the type, and the `proccode` consists of the complete code for the object subroutines of the record type. `moreproccode` is used for a dummy `a_MERGE` procedure definition if necessary, just like for the node types above.

270 *Connection Type Generation*[270] ≡

```
{
 RULE rConnectionTypeDef :
   ConnectionTypeDef ::=  'CONNECTION' ConElemDefList
 COMPUTE
   .datacode =        CONSTITUENTS ConDataElemDef.datacode
                      WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .descriptorcode =  PTGConTypeDescr();
   .initcode =        CONSTITUENTS ConDataElemDef.initcode
                      WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .printcode =       CONSTITUENTS ConDataElemDef.printcode
                      WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .proccode =        CONSTITUENTS (MergeProcDef.code, ObjProcedureDef.code,
                                    ObjFunctionDef.code)
                      WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   .moreproccode =
       IF (EQ (GetMergeDefs (INCLUDING TypeDefBody.InhKey, 0), 0),
         PTGMergeProcDef (INCLUDING TypeDefBody.InhPtg,
                          PTGStr ("/* dummy */")),
         PTGNULL) DEPENDS_ON ConnectionTypeDef.coded;
   ConnectionTypeDef.code =
     PTGSeq (
       PTGSeq ( /* typedefs: */
         PTGTypeDef (PTGSeq (INCLUDING TypeDefBody.InhPtg, PTGStr("_0")),
                     PTGRecordTypeDef (.datacode)),
         PTGTypeDef (INCLUDING TypeDefBody.InhPtg,
                     PTGConTypeDef (.datacode, .descriptorcode))),
       PTGSeq6 (
         PTGRecordTypeInit (INCLUDING TypeDefBody.InhPtg, .initcode),
         IF (ProducePrintcode,
           PTGConTypePrint (INCLUDING TypeDefBody.InhPtg, .printcode),
           PTGNULL),
         PTGConnectProcDef (INCLUDING TypeDefBody.InhPtg),
         PTGMergeConTpl (INCLUDING TypeDefBody.InhPtg),
         .moreproccode,
         .proccode));
   ConnectionTypeDef.coded = ORDER (
     SetSize (.Type, ConElemDefList.Offset, 0),
     SetAlignment (.Type, ConElemDefList.Alignment, 1));
   CHAINSTART ConElemDefList.Offset = 0;

 END;
```

```
RULE rConElemDefList1 :
  ConElemDefList ::=  ConElemDef ';'
COMPUTE
  ConElemDefList.Alignment = ConElemDef.Alignment;
END;

RULE rConElemDefList :
  ConElemDefList ::=  ConElemDefList ConElemDef ';'
COMPUTE
  ConElemDefList[1].Alignment =
    IF (GT (ConElemDefList[2].Alignment, ConElemDef.Alignment),
      ConElemDefList[2].Alignment, ConElemDef.Alignment);
END;

SYMBOL ConElemDef COMPUTE
  THIS.Alignment = 0;  /* for MERGE, FUNCTION, and PROCEDURE defs */
END;

RULE rConElemDef :
  ConElemDef ::=  ConDataElemDef
COMPUTE
  ConElemDef.Alignment = ConDataElemDef.Alignment;
END;

RULE rConDataElemDef :
  ConDataElemDef ::=  TypeId InitElemIdList
COMPUTE
  ConDataElemDef.datacode =
    PTGDataElemDef (TypeId.Ptg, InitElemIdList.datacode);
  ConDataElemDef.initcode = InitElemIdList.initcode;
  ConDataElemDef.printcode = InitElemIdList.printcode;
  ConDataElemDef.Alignment =
    GetAlignment (TypeId.Key, 1) DEPENDS_ON ConDataElemDef.coded;
  InitElemIdList.InhAlignment = ConDataElemDef.Alignment;
END;
}
```

This macro is defined in definitions 270.
This macro is invoked in definition 261.

44.5 Array types

Node arrays are represented as pointers (since they are allocated dynamically); other arrays are represented as plain MPL arrays. In both cases we also generate an initialization procedure.

Array Type PTG[271] ≡ 271

```
{
NodeArrayDef:   [IndentNewLine] "typedef plural " $2 "*  " $1 ";"
ArrayDef:       [IndentNewLine] "typedef " $2 "  " $1 "[" $3 "];"
NodeArrayTypeInitAlias:
   "\n#define INIT_" $1 "(a,b,c,d) INIT_" $2 "_group(a,b,c,d)\n"
NodeArrayTypePrint:
   "\nplural void p_pr" $1 " (String name, plural " $1 "* ME,"
   [IndentNewLine] "      plural _node_group_D* descr)"
   "\n{" [IndentIncr]
   [IndentNewLine] "char* n = \"0\";"
```

```
   [IndentNewLine] "int i;"
   [IndentNewLine] "printf (\"\\n%s@%x=(\", name, (int)ME);"
   [IndentNewLine] "p_pr_node_group_D (\"_D\", descr);"
   [IndentNewLine] "for (i = 0; i < descr->localsizeN; i++) {"
   [IndentNewLine] "  *n = (i % 64) + '0';"
   [IndentNewLine] "  p_pr" $2 " (n, (*ME)+i);"
   [IndentNewLine] "}"
   [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}\n"
ArrayTypeInit:
   "\n#define  _type_ " $1
   "\n#define  _basetype_ " $2
   "\n#define  _size_ " $3
   "\n#include \"ArrayInit.tpl\"\n"
ArrayTypePrint:
   "\nplural void p_pr" $1 " (String name, plural " $1 "* plural ME)"
   "\n{" [IndentIncr]
   [IndentNewLine] "int i;"
   [IndentNewLine] "printf (\"\\n%s@%x=(\", name, (int)_sgl(ME));"
   [IndentNewLine] "for (i = 0; i < " $3 "; i++) {"
   [IndentNewLine] "  printf (\"%d\", i);"
   [IndentNewLine] "  p_pr" $2 " (\"\", (*ME)+i);"
   [IndentNewLine] "}"
   [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}\n"
}
```

This macro is defined in definitions 271.
This macro is invoked in definition 260.

272 *Array Type Generation*[272] ≡

```
{
RULE rArrayTypeDef :
  ArrayTypeDef ::=  'ARRAY' '[' ArraySize ']' 'OF' TypeId
COMPUTE
  .datacode =
     IF (EQ (TypeId.Kind, NodeTypeK),
       PTGNodeArrayDef (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg),
       PTGArrayDef (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg,
                    PTGStr (Const2Str (ArraySize.Val))));
  .initcode =
     IF (EQ (TypeId.Kind, NodeTypeK),
       PTGNodeArrayTypeInitAlias (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg),
     IF (EQ (TypeId.Kind, RecordTypeK),
       PTGArrayTypeInit (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg,
                         PTGInt (GetIval (ArraySize.Val))),
     /* else */
       PTGNULL));
  .printcode =
     IF (EQ (TypeId.Kind, NodeTypeK),
       PTGNodeArrayTypePrint (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg),
     /* else */
     IF (EQ (TypeId.Kind, RecordTypeK),
       PTGArrayTypePrint (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg,
                          PTGInt (GetIval (ArraySize.Val))),
     /* else */
       PTGNULL));
  ArrayTypeDef.code = PTGSeq3 (.datacode, .initcode,
                               IF (ProducePrintcode, .printcode, PTGNULL));
```

```
  ArrayTypeDef.coded = ORDER (
    SetSize (INCLUDING TypeDefBody.InhKey,
             MUL(GetIval(ArraySize.Val),GetSize(TypeId.Key,0)), 0),
    SetAlignment (INCLUDING TypeDefBody.InhKey,
                  GetAlignment (TypeId.Key, 1), 1))
    DEPENDS_ON ArrayTypeDef.coded;

END;
}
```

This macro is defined in definitions 272.
This macro is invoked in definition 261.

44.6 Group types

Group types are handled just like node arrays of size 0.

Group Type PTG[273] ≡ 273

```
{
/* nothing special needed for GROUPS */
}
```

This macro is defined in definitions 273.
This macro is invoked in definition 260.

Group Type Generation[274] ≡ 274

```
{
RULE rGroupTypeDef :
  GroupTypeDef ::=  'GROUP' 'OF' TypeId
COMPUTE
  .datacode = PTGNodeArrayDef (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg);
  .initcode = PTGNodeArrayTypeInitAlias (INCLUDING TypeDefBody.InhPtg,
                                         TypeId.Ptg);
  .printcode = PTGNodeArrayTypePrint (INCLUDING TypeDefBody.InhPtg, TypeId.Ptg);
   GroupTypeDef.code = PTGSeq3 (.datacode, .initcode,
                                IF (ProducePrintcode, .printcode, PTGNULL));
END;
}
```

This macro is defined in definitions 274.
This macro is invoked in definition 261.

44.7 Network types

Network Type PTG[275] ≡ 275

```
{
NetType: "struct { /* NETWORK */" [IndentIncr]
         $1
         [IndentNewLine] "/* Node groups: */" $2
         [IndentNewLine] "/* Descriptors: */" $3
         [IndentNewLine] "_network_D _me_D;"
         [IndentDecr] [IndentNewLine] "} "
NetTypeInit: "\nplural void INIT_" $1 " (plural " $1 "* ME)"
             "\n{" [IndentIncr] $2 [IndentDecr] "\n}\n"
NetInit: [IndentNewLine] "ME->_me_D.exists = _existing;"
         [IndentNewLine] "ME->_me_D.formA = false;"
         [IndentNewLine] "ME->_me_D.meI   = 0;"
```

```
          [IndentNewLine] "ME->_me_D.repN  = 1;"
          [IndentNewLine] "ME->_me_D.lrepN = 0;"
          [IndentNewLine] "ME->_me_D.lxN   = lxprocN;"
          [IndentNewLine] "ME->_me_D.lyN   = lyprocN;"
                           $1/*data element init*/
NetTypePrint:
          "\nplural void p_pr" $1 " (String name, plural " $1 "* ME)"
          "\n{" [IndentIncr]
          [IndentNewLine] "printf (\"\\n%s@%x=(\", name, (int)ME);"
          [IndentNewLine] "p_pr_network_D (\"_me_D\", &ME->_me_D);"
          [IndentNewLine] "if (ME->_me_D.exists) {"
          [IndentIncr]       $2   [IndentDecr]
          [IndentNewLine] "}"
          [IndentNewLine] "printf (\")\");" [IndentDecr] "\n}\n"
MergeNetTpl:  "\n#define _type_ " $1
              "\n#define _GROUPS_ " $2
              "\n#include \"MergeNet.tpl\"\n"
ReplicateNetTpl:  "\n#define _type_ " $1
                  "\n#define _GROUPS_ " $2
                  "\n#include \"ReplicateNet.tpl\"\n"
}
```

This macro is defined in definitions 275.
This macro is invoked in definition 260.

The code generation for network types is structured along the same lines as that of record, node, and connection types.

276 *Network Type Generation*[276] ≡

```
{
RULE rNetworkTypeDef :
  NetworkTypeDef ::=  'NETWORK' NetElemDefList
COMPUTE
  .datacode =       CONSTITUENTS NetDataElemDef.datacode
                    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .moredatacode =   CONSTITUENTS NetDataElemDef.moredatacode
                    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .descriptorcode = CONSTITUENTS NetDataElemDef.descriptorcode
                    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .initcode =       CONSTITUENTS NetDataElemDef.initcode
                    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .printcode =      CONSTITUENTS NetDataElemDef.printcode
                    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .proccode =       CONSTITUENTS (MergeProcDef.code, ObjProcedureDef.code,
                                  ObjFunctionDef.code)
                    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
  .moreproccode =
      IF (EQ (GetMergeDefs (INCLUDING TypeDefBody.InhKey, 0), 0),
        PTGMergeProcDef (INCLUDING TypeDefBody.InhPtg,
                         PTGStr ("/* dummy */")),
        PTGNULL) DEPENDS_ON NetworkTypeDef.coded;
  NetworkTypeDef.code =
    PTGSeq (
      PTGSeq ( /* typedefs: */
        PTGTypeDef (PTGSeq (INCLUDING TypeDefBody.InhPtg, PTGStr("_0")),
                    PTGRecordTypeDef (.datacode)),
        PTGTypeDef (INCLUDING TypeDefBody.InhPtg,
                    PTGNetType (.datacode, .moredatacode, .descriptorcode))),
```

```
        PTGSeq6 ( /* functions: */
          PTGNetTypeInit (INCLUDING TypeDefBody.InhPtg, PTGNetInit(.initcode)),
          IF (ProducePrintcode,
            PTGNetTypePrint (INCLUDING TypeDefBody.InhPtg, .printcode),
            PTGNULL),
          PTGMergeNetTpl (INCLUDING TypeDefBody.InhPtg, NetworkTypeDef.Groups),
          .moreproccode,
          PTGReplicateNetTpl (INCLUDING TypeDefBody.InhPtg,
                              NetworkTypeDef.Groups),
          .proccode));
    NetworkTypeDef.Groups = CONSTITUENTS (InitElemId.Groups)
                            WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
    /* dummy CHAINSTART (for InitElemId), chain is not actually needed: */
    CHAINSTART NetElemDefList.Offset = 0;
  END;

  RULE rNetDataElemDef :
    NetDataElemDef ::=  TypeId InitElemIdList
  COMPUTE
    .code = PTGDataElemDef (TypeId.Ptg, InitElemIdList.datacode);
    NetDataElemDef.datacode =
      IF (OR (EQ (TypeId.Kind, NodeArrayTypeK),
              EQ (TypeId.Kind, NodeGroupTypeK)),
        PTGNULL, /* else */ .code);
    NetDataElemDef.moredatacode =
      IF (OR (EQ (TypeId.Kind, NodeArrayTypeK),
              EQ (TypeId.Kind, NodeGroupTypeK)),
        .code, /* else */ PTGNULL);
    NetDataElemDef.descriptorcode = InitElemIdList.descriptorcode;
    NetDataElemDef.initcode = InitElemIdList.initcode;
    NetDataElemDef.printcode = InitElemIdList.printcode;
    InitElemIdList.InhAlignment = 1; /* dummy to satisfy InitElemIdList */
  END;
  }
```

This macro is defined in definitions 276.
This macro is invoked in definition 261.

45 Data object definitions

For each object in a definition list, we generate a complete MPL declaration.

Data Object Definition PTG[277] ≡ 277

```
 {
  VarDef:     [IndentNewLine] $1/*type*/ "  " $2/*name*/ $3/*initialization*/ ";"
  ParVarDef:  [IndentNewLine] "plural " $1 "  " $2 $3 ";"
  IoDef:      [IndentNewLine] "plural " $1 "  *" $2 ";"
  }
```

This macro is defined in definitions 277.
This macro is invoked in definition 336.

Data objects have to be generated in different forms for sequential and parallel contexts. For each object in a CuPit declaration list we generate an individual MPL declaration since this is a bit easier. I/O objects are implemented as pointers.

Data Object Definition Generation[278] ≡ 278

```
{
 RULE rDataObjectDef:
   DataObjectDef ::=  TypeId AccessType InitDataIdList
 COMPUTE
   InitDataIdList.InhPtg = TypeId.Ptg;
   DataObjectDef.seqcode = CONSTITUENTS InitDataId.seqcode
                           WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   DataObjectDef.parcode = CONSTITUENTS InitDataId.parcode
                           WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
   DataObjectDef.code = /* used for global defs only */
     IF (EQ (TypeId.Kind, NetTypeK),
       DataObjectDef.parcode, /* else */ DataObjectDef.seqcode);
 END;
 }
```

This macro is defined in definitions 278, 279, 280, and 281.
This macro is invoked in definition 334.

279 *Data Object Definition Generation*[279] ≡

```
{
 RULE rInitDataIdList :
   InitDataIdList ::=  InitDataIdList ',' InitDataId
 COMPUTE
   TRANSFER InhPtg;
 END;
 }
```

This macro is defined in definitions 278, 279, 280, and 281.
This macro is invoked in definition 334.

Data identifiers without an initializer may be `VAR` or `IO` in sequential context, but only `VAR` in parallel context.

280 *Data Object Definition Generation*[280] ≡

```
{
 RULE rInitDataId0 :
   InitDataId ::=  NewDataId
 COMPUTE
   InitDataId.seqcode =
     IF (IsVarAcc (INCLUDING InitDataIdList.InhAccess),
       PTGVarDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg, PTGNULL),
     /* else */
     IF (IsIoAcc (INCLUDING InitDataIdList.InhAccess),
       PTGIoDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg),
     /* else */
       PTGStr ("/* ERROR: Illegal access value !!! */")));
   InitDataId.parcode =
     IF (IsVarAcc (INCLUDING InitDataIdList.InhAccess),
       PTGParVarDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg, PTGNULL),
       PTGStr ("/* ERROR: Illegal access value !!! */"));
 END;
 }
```

This macro is defined in definitions 278, 279, 280, and 281.
This macro is invoked in definition 334.

Data identifiers with an initializer may be `VAR` or `CONST` in either sequential or parallel context. They are currently handled in exactly the same way.

281 *Data Object Definition Generation*[281] ≡

```
{
```

```
RULE rInitDataId1 :
  InitDataId ::=  NewDataId ':=' Expr
COMPUTE
  InitDataId.seqcode =
    IF (IsVarAcc (INCLUDING InitDataIdList.InhAccess),
      PTGVarDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg,
                 PTGSeq (PTGStr (" = "), Expr.seqcode)),
    /* else */
    IF (IsConstAcc (INCLUDING InitDataIdList.InhAccess),
      PTGVarDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg,
                 PTGSeq (PTGStr (" = "), Expr.seqcode)),
    /* else */
      PTGStr ("/* ERROR: Illegal access value !!! */")));
  InitDataId.parcode =
    IF (IsVarAcc (INCLUDING InitDataIdList.InhAccess),
      PTGParVarDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg,
                    PTGSeq (PTGStr (" = "), Expr.parcode)),
    /* else */
    IF (IsConstAcc (INCLUDING InitDataIdList.InhAccess),
      PTGParVarDef (INCLUDING InitDataIdList.InhPtg, NewDataId.Ptg,
                    PTGSeq (PTGStr (" = "), Expr.parcode)),
    /* else */
      PTGStr ("/* ERROR: Illegal access value !!! */")));
END;
}
```

This macro is defined in definitions 278, 279, 280, and 281.
This macro is invoked in definition 334.

46 Subroutine definitions

Free global subroutines and object subroutines of RECORD types are generated in a sequential as well as a parallel version. Global subroutines that belong to the central agent are generated in a sequential version only. Object subroutines of CONNECTION, NODE, and NETWORK types as well as MERGE procedures, REDUCTION functions, and WTA functions are generated in parallel versions only — but two of them: a direct implementation of the CuPit procedure or function as a straightforward plural MPL function plus an additional wrapper function that implements the virtualization.

Subroutine Definition PTG[282] ≡ 282

```
{
SubroutineDef:   [IndentNewLine] $1/*type*/ " " $2/*name*/ $3/*params&body*/
SubroutineExt:   " (" [IndentIncr] $1 [IndentDecr] ");"
SubroutineDescr: " (" [IndentIncr] $1 [IndentDecr] ")"
                 [IndentNewLine] "{" [IndentIncr] $2 [IndentDecr]
                 [IndentNewLine] "}" [IndentNewLine]
aObjProcedureDef:
   [IndentNewLine] "plural void a_" $1 "_" $2 " ("
                   "plural " $2 "* ME," [IndentIncr]
   [IndentNewLine] "_sint _localsize, plural Interval slice"
                   $3 ")" [IndentDecr]
   [IndentNewLine] "{" [IndentIncr]
   [IndentNewLine] "_sint _i;"
   [IndentNewLine] "plural Int _first = slice.min, _last = slice.max;"
   [IndentNewLine] "_TRACE (3, (\"a_" $1 "_" $2 " (%x %d...%d)\\n\", (int)ME, "
                      "_sgl(_first), _sgl(_last)));"
```

```
      [IndentNewLine] "for (_i = 0; _i < _localsize; _i++, ME++)" [IndentIncr]
      [IndentNewLine] "if (ME->_me_D.exists && "
      [IndentNewLine] "    ME->_me_D.meI >=_first && ME->_me_D.meI <=_last)"
      [IndentNewLine] "  " $1 "_" $2 " (" $4 ");" [IndentDecr] [IndentDecr]
      [IndentNewLine] "}" [IndentNewLine]
   ObjSubroutineName:   $1 "_" $2
   ObjMeDef:         "plural " $1 "* ME"
   ParName:          "p_" $1
   Paramlist:        $1 "," [IndentNewLine] $2
   VarParam:         $1 " *" $2
   ConstParam:       $1 " " $2
   IoParam:          "plural " $1 " **" $2
   ParVarParam:      "plural " $1 "* plural " $2
   ParConstParam:    "plural " $1 " " $2
   ParIoParam:       "plural " $1 " **" $2
   }
```

This macro is defined in definitions 282, 283, 284, 288, 290, and 292.
This macro is invoked in definition 336.

The virtualized local connection procedures have to postprocess each call to the non-virtualized connection procedure: Should the connection delete itself during the call, the value of the connections' remote end `exists` indicators must be changed and the `conN` values of the interface descriptors be invalidated. This is done by a call to the procedure `delete_connection_postprocessing`.

283 *Subroutine Definition PTG*[283] ≡

```
  {
   aConProcedureDef:
      /* arguments: $1=procedure name, $2=type name, $3=parameter list,
                    $4=argument list,  $5=dummy send/fetch-code (if any)
      */
      [IndentNewLine] "plural void a_" $1 "_" $2 " (" [IndentIncr]
      [IndentNewLine] "register plural " $2 "* ME, register _sint _localsize"
                      $3 ")" [IndentDecr]
      [IndentNewLine] "{" [IndentIncr]
      [IndentNewLine] "register int _i;"
      [IndentNewLine] "_TRACE (4, (\"a_" $1 "_" $2 " (%x)\\n\", (int)ME));"
      [IndentNewLine] "dpuTimerStart ();"
      [IndentNewLine] "for (_i = 0; _i < _localsize; _i++, ME++)" [IndentIncr]
      [IndentNewLine] "if (ME->_me_D.exists) {" [IndentIncr]
      [IndentNewLine] $1 "_" $2 " (" $4 ");"
                      $5
      [IndentNewLine] "if (!ME->_me_D.exists) {" [IndentIncr]
      [IndentNewLine] "delete_connection_postprocessing (&ME->_me_D, &ME->_oe,"
      [IndentNewLine] "   offsetof(_remote_connection,_me_D));"  [IndentDecr]
      [IndentNewLine] "}" [IndentDecr]
      [IndentNewLine] "}" [IndentDecr]
                      "\n#if _codetype_ == 1"
      [IndentNewLine] "ME -= _localsize;"
      [IndentNewLine] "if (_localsize && ME->_me_D.exists && "
                           "ME->_me_D.boss->boss->boss->boss->formA) {"
      [IndentNewLine] "  unsigned ticks = dpuTimerTicks2() - 300;"
      [IndentNewLine] "  ME->_me_D.boss->wpc += ticks/_localsize;"
      [IndentNewLine] "}"
                      "\n#endif" [IndentDecr]
      [IndentNewLine] "}" [IndentNewLine]
   dummySendFetchCode:
      [IndentNewLine] "{ plural " $3 " _buf; plural " $3 "* _ME = ME, *ME = &_buf;"
```

```
                        $1 $2 " }"
    }
```

This macro is defined in definitions 282, 283, 284, 288, 290, and 292.
This macro is invoked in definition 336.

The virtualized remote connection procedures have to pre- and postprocess each call to the non-virtualized connection procedure: Before the call, an appropriate local connection object has to be constructed. To do this, all relevant (i.e. may-be-read) parts of the connection object have to be fetched (done by the code `$5`). After the call there are two possibilities: Should the connection have deleted itself during the call, the `exists` indicators and interface descriptor's `conN` values must be changed at both ends of the connection (using `delete_connection_postprocessing` just like for the local case above). Should the connection still exist after the call, those parts of it that (may) have changed must be written back to the data end of the connection (done by code `$6`).

Subroutine Definition PTG[284] ≡ 284

```
  {
   arConProcedureDef:
      [IndentNewLine] "plural void a_" $1 "_remote_" $2 " (" [IndentIncr]
      [IndentNewLine] "register plural _remote_connection* _ME, "
                      "register _sint _localsize"
                      $3 ")" [IndentDecr]
      [IndentNewLine] "{" [IndentIncr]
      [IndentNewLine] "register int _i;  _work cost = 0;"
      [IndentNewLine] "plural " $2 "  _buffer;"
      [IndentNewLine] "register plural " $2 " *ME = &_buffer;"
      [IndentNewLine] "_TRACE (4, (\"a_" $1 "_remote_" $2 " (%x)\\n\", (int)_ME));"
      [IndentNewLine] "for (_i = 0; _i < _localsize; _i++, _ME++)" [IndentIncr]
      [IndentNewLine] "if (_ME->_me_D.exists) {" [IndentIncr]
                      $5 /* fetch */
      [IndentNewLine] "ME->_me_D.exists = true;"
      [IndentNewLine] "dpuTimerStart ();"
      [IndentNewLine] $1 "_" $2 " (" $4 ");"
                      "\n#if _codetype_ == 1"
      [IndentNewLine] "cost += dpuTimerTicks2();"
                      "\n#endif"
      [IndentNewLine] "if (ME->_me_D.exists) {" [IndentIncr]
                      $6 /* send */     [IndentDecr]
      [IndentNewLine] "}"
      [IndentNewLine] "else {" [IndentIncr]
      [IndentNewLine] "delete_connection_postprocessing (&_ME->_me_D, &_ME->_oe,"
      [IndentNewLine] "   offsetof(" $2 ",_me_D));"  [IndentDecr]
      [IndentNewLine] "}" [IndentDecr]
      [IndentNewLine] "}" [IndentDecr]
                      "\n#if _codetype_ == 1"
      [IndentNewLine] "_ME -= _localsize;"
      [IndentNewLine] "if (_localsize && _ME->_me_D.exists && "
                           "_ME->_me_D.boss->boss->boss->boss->formA) {"
      [IndentNewLine] "  _ME->_me_D.boss->wpc += cost/_localsize-300+" $7 ";"
      [IndentNewLine] "}"
                      "\n#endif" [IndentDecr]
      [IndentNewLine] "}" [IndentNewLine]
  }
```

This macro is defined in definitions 282, 283, 284, 288, 290, and 292.
This macro is invoked in definition 336.

46.1 Procedures and functions

For global functions and free global procedures we generate a sequential and a parallel version. For procedures of the central agent, we generate a sequential version only. No virtualization wrapper for the parallel version is needed because free parallel subroutines can only be called from (connection, node, or network) contexts that are already virtualized but they can never themselves introduce additional parallelism that could need virtualization. For unused procedures, no code is generated; individual versions (sequential/parallel or parallel/virtualized/remote) are suppressed if possible.

285 *Subroutine Definition Generation*[285] ≡

```
{
 RULE rProcedureDef :
   ProcedureDef ::=  'PROCEDURE' NewProcedureId SubroutineDescription
                     OptPROCEDURE
 COMPUTE
   ProcedureDef.code =
     PTGSeq (
       IF (OR (ProcedureDef.CentralAgent,
           EQ (BITAND (GetIsUsed (NewProcedureId.Key, used0), used), used)),
         PTGSubroutineDef (PTGStr ("void"), NewProcedureId.Ptg,
                          SubroutineDescription.seqcode),
         PTGNULL),  /* no sequential version */
       IF (OR (ProcedureDef.CentralAgent,
               NE (BITAND (GetIsUsed(NewProcedureId.Key,used0), usedA), usedA)),
         PTGNULL,   /* no parallel version */
         PTGSubroutineDef (PTGStr ("plural void"),
                           PTGParName (NewProcedureId.Ptg),
                           SubroutineDescription.parcode)))
     DEPENDS_ON INCLUDING CupitProgram.allknown;
   SubroutineDescription.InhPtg = PTGNULL; /* no ME type name */
 END;

 RULE rFunctionDef :
   FunctionDef ::=  TypeId 'FUNCTION' NewFunctionId SubroutineDescription
                    OptFUNCTION
 COMPUTE
   FunctionDef.code =
     PTGSeq (
       IF (OR (FunctionDef.CentralAgent,
           EQ (BITAND (GetIsUsed (NewFunctionId.Key, used0), used), used)),
         PTGSubroutineDef (TypeId.Ptg, NewFunctionId.Ptg,
                           SubroutineDescription.seqcode),
         PTGNULL),  /* no sequential version */
       IF (OR (FunctionDef.CentralAgent,
               NE (BITAND (GetIsUsed(NewFunctionId.Key,used0), usedA), usedA)),
         PTGNULL,   /* no parallel version */
         PTGSubroutineDef (PTGSeq (PTGStr ("plural "), TypeId.Ptg),
                           PTGParName (NewFunctionId.Ptg),
                           SubroutineDescription.parcode)))
     DEPENDS_ON INCLUDING CupitProgram.allknown;
   SubroutineDescription.InhPtg = PTGNULL;  /* no ME type name */
 END;
}
```

This macro is defined in definitions 285, 286, 287, 289, 291, and 293.
This macro is invoked in definition 334.

For object functions, we generate a simple parallel version only, because it is impossible to call a function that introduces additional parallelism. For object procedures, we generate (1) a parallel version, (2) a virtualized parallel version (unless the object type is a network type), and perhaps (3) a remote virtualized version (if the object type is a connection type).

Subroutine Definition Generation[286] ≡ 286

```
{
 RULE rObjProcedureDef :
   ObjProcedureDef ::=  'PROCEDURE' NewObjProcedureId SubroutineDescription
                        OptPROCEDURE
 COMPUTE
   .parcode = /* the non-virtualized parallel version */
       IF (EQ (GetIsUsed(NewObjProcedureId.Key,used0), used0),
         PTGNULL,  /* is an unused object procedure */
         PTGSubroutineDef (PTGStr ("plural void"),
             PTGObjSubroutineName (NewObjProcedureId.Ptg,
                                   INCLUDING TypeDefBody.InhPtg),
             SubroutineDescription.parcode))
       DEPENDS_ON INCLUDING CupitProgram.allknown;
   .sendcode =
       IF (NE (.Kind, ConTypeK),
         PTGNULL,
         ORDER (makeRemoteSendCode (
                  GetMayBeWritten (NewObjProcedureId.Key, NoDefTblKeySet),
                  OR (conIndividual, conWhole),
                  IF (conIndividual, NoKey, INCLUDING TypeDefBody.InhKey),
                  hideLatency, highLatency),
                getRemoteSendCode ())) DEPENDS_ON ObjProcedureDef.coded;
   .fetchcode =
       IF (NE (.Kind, ConTypeK),
         PTGNULL,
         ORDER (makeRemoteFetchCode (
                  DSunite (GetMayBeRead (NewObjProcedureId.Key, NoDefTblKeySet),
                       GetMayBeWritten (NewObjProcedureId.Key, NoDefTblKeySet)),
                  OR (conIndividual, conWhole),
                  IF (conIndividual, NoKey, INCLUDING TypeDefBody.InhKey),
                  hideLatency, highLatency),
                getRemoteFetchCode ())) DEPENDS_ON ObjProcedureDef.coded;
   .rticode = getRemoteCommCost () DEPENDS_ON (.sendcode, .fetchcode);
   .smallcode = /* the virtualized a_-version, if needed */
     IF (EQ (.Kind, ConTypeK),
       IF (NE (BITAND (GetIsUsed(NewObjProcedureId.Key,used0), usedA), usedA),
         PTGNULL, /* virtualized connection procedure never used */
         PTGaConProcedureDef (NewObjProcedureId.Ptg,
           INCLUDING TypeDefBody.InhPtg,
           WithComma (SubroutineDescription.paramlist),
           SubroutineDescription.arglist,
           IF (noDataLocality,
               PTGdummySendFetchCode (.fetchcode, .sendcode,
                                      INCLUDING TypeDefBody.InhPtg),
               PTGNULL))),
     /* else */
     IF (EQ (.Kind, NodeTypeK),
       IF (NE (BITAND (GetIsUsed(NewObjProcedureId.Key,used0), usedA), usedA),
         PTGNULL, /* virtualized node procedure never used */
         PTGaObjProcedureDef (NewObjProcedureId.Ptg,
           INCLUDING TypeDefBody.InhPtg,
```

```
          WithComma (SubroutineDescription.paramlist),
          SubroutineDescription.arglist)),
    /* else */
      PTGNULL))  /* no virtualization for NETWORK or RECORD objects */
    DEPENDS_ON INCLUDING CupitProgram.allknown;
  .largecode = /* the virtualized a_remote-version, if needed */
    IF (OR (NE (.Kind, ConTypeK),
        NE (BITAND (GetIsUsed(NewObjProcedureId.Key,used0), usedAR), usedAR)),
      PTGNULL,  /* remote procedures only for CONNECTIONs, where used */
      PTGarConProcedureDef (NewObjProcedureId.Ptg,
          INCLUDING TypeDefBody.InhPtg,
          WithComma (SubroutineDescription.paramlist),
          SubroutineDescription.arglist,
          .fetchcode, .sendcode, .rticode))
     DEPENDS_ON ObjProcedureDef.coded;
  ObjProcedureDef.code = PTGSeq3 (.parcode, .smallcode, .largecode);
  ObjProcedureDef.coded = ObjProcedureDef.code;
  SubroutineDescription.InhPtg = INCLUDING TypeDefBody.InhPtg;  /* ME type */
END;

RULE rObjFunctionDef :
  ObjFunctionDef ::=  TypeId 'FUNCTION' NewObjFunctionId SubroutineDescription
                      OptFUNCTION
COMPUTE
  ObjFunctionDef.code =
    IF (EQ (GetIsUsed (NewObjFunctionId.Key, used0), used0),
      PTGNULL, /* function is not used */
      PTGSubroutineDef (PTGSeq (PTGStr ("plural "), TypeId.Ptg),
          PTGObjSubroutineName (NewObjFunctionId.Ptg,
                                INCLUDING TypeDefBody.InhPtg),
          SubroutineDescription.parcode));
  SubroutineDescription.InhPtg = INCLUDING TypeDefBody.InhPtg;  /* ME type */
END;
}
```

This macro is defined in definitions 285, 286, 287, 289, 291, and 293.
This macro is invoked in definition 334.

46.2 Parameter lists

The now following parameter list handling is no doubt long, but also relatively regular in its structure. (I suggest that you skip this section or otherwise you might fall asleep.)

287 *Subroutine Definition Generation*[287] ≡

```
{
RULE rSubroutineDescription :
  SubroutineDescription ::=  ParamList 'IS' SubroutineBody 'END'
COMPUTE
  .parcode = /* ParamList with ME prepended if necessary */
      IF (EQ (SubroutineDescription.Context, ObjSubroutineContext),
        IF (EQ (ParamList.parcode, PTGNULL),  /* no parameters ? */
          PTGObjMeDef (SubroutineDescription.InhPtg), /* then ME only */
          PTGParamlist (PTGObjMeDef (SubroutineDescription.InhPtg),
                        ParamList.parcode)),          /* else ME + others */
        ParamList.parcode);  /* or pure paramlist for non-object subroutines */
  SubroutineDescription.seqcode =
```

```
     PTGSubroutineDescr (ParamList.seqcode, SubroutineBody.seqcode);
  SubroutineDescription.parcode =
    PTGSubroutineDescr (.parcode, SubroutineBody.parcode);
  SubroutineDescription.paramlist = ParamList.parcode;
  SubroutineDescription.arglist   = ParamList.arglist;
END;

RULE rSubroutineDescription0 :
  SubroutineDescription ::=  ParamList 'IS' 'EXTERNAL'
COMPUTE
  .parcode = /* ParamList with ME prepended if necessary */
      IF (EQ (SubroutineDescription.Context, ObjSubroutineContext),
        PTGParamlist (PTGObjMeDef (SubroutineDescription.InhPtg),
                      ParamList.parcode),
        ParamList.parcode);
  SubroutineDescription.seqcode = PTGSubroutineExt (ParamList.seqcode);
  SubroutineDescription.parcode = PTGSubroutineExt (.parcode);
  SubroutineDescription.paramlist = ParamList.parcode;
  SubroutineDescription.arglist   = ParamList.arglist;
END;

RULE rParamList0 :
  ParamList ::=  '(' ')'
COMPUTE
  ParamList.seqcode = PTGNULL; /* nothing needed */
  ParamList.parcode = PTGNULL;
  ParamList.arglist =
    IF (EQ (INCLUDING SubroutineDescription.Context, ObjSubroutineContext),
      PTGStr ("ME"), /* else */ PTGNULL);
END;

RULE rParamList :
  ParamList ::=  '(' Parameters ')'
COMPUTE
  ParamList.seqcode = Parameters.seqcode;
  ParamList.parcode = Parameters.parcode;
  ParamList.arglist =
    IF (EQ (INCLUDING SubroutineDescription.Context, ObjSubroutineContext),
      PTGList (PTGStr ("ME"), Parameters.arglist),
      Parameters.arglist);
END;

RULE rParameters1 :
  Parameters ::=  ParamsDef
COMPUTE
  Parameters.seqcode = ParamsDef.seqcode;
  Parameters.parcode = ParamsDef.parcode;
  Parameters.arglist = ParamsDef.arglist;
END;

RULE rParameters :
  Parameters ::=  Parameters ';' ParamsDef
COMPUTE
  Parameters[1].seqcode =
    PTGParamlist (Parameters[2].seqcode, ParamsDef.seqcode);
```

```
  Parameters[1].parcode =
    PTGParamlist (Parameters[2].parcode, ParamsDef.parcode);
  Parameters[1].arglist = PTGList (Parameters[2].arglist, ParamsDef.arglist);
END;

RULE rParamsDef :
  ParamsDef ::=  TypeId AccessType ParamIdList
COMPUTE
  ParamIdList.InhPtg = TypeId.Ptg;
  ParamsDef.seqcode = ParamIdList.seqcode;
  ParamsDef.parcode = ParamIdList.parcode;
  ParamsDef.arglist = ParamIdList.arglist;
END;

RULE rParamIdList1 :
  ParamIdList ::=  NewParamId
COMPUTE
  ParamIdList.seqcode =
    IF (IsVarAcc (ParamIdList.InhAccess),
      PTGVarParam (ParamIdList.InhPtg, NewParamId.Ptg),
    /* else */
    IF (IsConstAcc (ParamIdList.InhAccess),
      PTGConstParam (ParamIdList.InhPtg, NewParamId.Ptg),
    /* else */
    IF (IsIoAcc (ParamIdList.InhAccess),
      PTGIoParam (ParamIdList.InhPtg, NewParamId.Ptg),
    /* else */
      PTGStr ("/* ERROR: Illegal access value !!! */"))));
  ParamIdList.parcode =
    IF (IsVarAcc (ParamIdList.InhAccess),
      PTGParVarParam (ParamIdList.InhPtg, NewParamId.Ptg),
    /* else */
    IF (IsConstAcc (ParamIdList.InhAccess),
      PTGParConstParam (ParamIdList.InhPtg, NewParamId.Ptg),
    /* else */
    IF (IsIoAcc (ParamIdList.InhAccess),
      PTGParIoParam (ParamIdList.InhPtg, NewParamId.Ptg),
    /* else */
      PTGStr ("/* ERROR: Illegal access value !!! */"))));
  ParamIdList.arglist = NewParamId.Ptg;
END;

RULE rParamIdList :
  ParamIdList ::=  ParamIdList ',' NewParamId
COMPUTE
  TRANSFER InhPtg WITH ParamIdList[2];
  ParamIdList[1].seqcode =
    IF (IsVarAcc (ParamIdList[2].InhAccess),
      PTGParamlist (ParamIdList[2].seqcode,
                    PTGVarParam (ParamIdList[2].InhPtg, NewParamId.Ptg)),
    /* else */
    IF (IsConstAcc (ParamIdList[2].InhAccess),
      PTGParamlist (ParamIdList[2].seqcode,
                    PTGConstParam (ParamIdList[2].InhPtg, NewParamId.Ptg)),
    /* else */
```

```
    IF (IsIoAcc (ParamIdList[2].InhAccess),
      PTGParamlist (ParamIdList[2].seqcode,
                    PTGIoParam (ParamIdList[2].InhPtg, NewParamId.Ptg)),
    /* else */
      PTGStr ("/* ERROR: Illegal access value !!! */"))));
  ParamIdList[1].parcode =
    IF (IsVarAcc (ParamIdList[2].InhAccess),
      PTGParamlist (ParamIdList[2].parcode,
                    PTGParVarParam (ParamIdList[2].InhPtg, NewParamId.Ptg)),
    /* else */
    IF (IsConstAcc (ParamIdList[2].InhAccess),
      PTGParamlist (ParamIdList[2].parcode,
                    PTGParConstParam (ParamIdList[2].InhPtg, NewParamId.Ptg)),
    /* else */
    IF (IsIoAcc (ParamIdList[2].InhAccess),
      PTGParamlist (ParamIdList[2].parcode,
                    PTGParIoParam (ParamIdList[2].InhPtg, NewParamId.Ptg)),
    /* else */
      PTGStr ("/* ERROR: Illegal access value !!! */"))));
  ParamIdList[1].arglist = PTGList (ParamIdList[2].arglist, NewParamId.Ptg);
END;

RULE rSubroutineBody :
  SubroutineBody ::=  Statements
COMPUTE
  TRANSFER seqcode, parcode;
END;
}
```

This macro is defined in definitions 285, 286, 287, 289, 291, and 293.
This macro is invoked in definition 334.

46.3 Reduction functions

For reduction functions, we have to generate two parts of code. First, the reduction function (implementing the actual reduction operator) itself; this is handled much like an object function. Second, the corresponding virtualization procedure(s); these have to be implemented individually for connections (if needed), nodes (if needed), and networks (if needed), but each of these three kinds may be implemented only once for each reduction function result type, independent of how many reduction functions with the same return type exist. The virtualization procedures are implemented as templates.

Subroutine Definition PTG[288] ≡ 288

```
{
ReductionFunctionDef:  [IndentNewLine] "plural " $1 " p_" $2
                 " (plural " $1 "* ME, plural " $1 "* YOU)"
                 [IndentNewLine] "{" [IndentIncr]
                 $3 [IndentDecr]
                 [IndentNewLine] "}" [IndentNewLine]
ReductionTplDef: [IndentNewLine] [IndentNewLine] "#define _type_ " $1
                 [IndentNewLine] "#include \"Reduction" $2 ".tpl\""
                 [IndentNewLine] [IndentNewLine]
}
```

This macro is defined in definitions 282, 283, 284, 288, 290, and 292.
This macro is invoked in definition 336.

Subroutine Definition Generation[289] ≡ 289

```
{
 RULE rReductionFunctionDef :
   ReductionFunctionDef ::=  TypeId 'REDUCTION' NewReductionFunctionId 'IS'
                             ReductionFunctionBody 'END' OptREDUCTION
 COMPUTE
   .smallcode = /* the reduction function itself */
      PTGReductionFunctionDef (TypeId.Ptg, NewReductionFunctionId.Ptg,
                               ReductionFunctionBody.parcode);
   .largecode = /* the a_REDUCTION function(s) (if needed) */
      PTGSeq3 (
        IF (GetNeedConReduction (TypeId.Key, false),
          ORDER (Messag1 (NOTE, "NeedConReduction(%s)=true",
                          SymString (TypeId.Sym)),
                 SetNeedConReduction (TypeId.Key, false, false),
                 PTGReductionTplDef(TypeId.Ptg, PTGStr ("Con"))),
          ORDER (Messag1 (NOTE, "NeedConReduction(%s)=false",
                          SymString (TypeId.Sym)),
                 PTGNULL)),
        IF (GetNeedNodeReduction (TypeId.Key, false),
          ORDER (SetNeedNodeReduction (TypeId.Key, false, false),
                 PTGReductionTplDef(TypeId.Ptg, PTGStr ("Node"))),
          PTGNULL),
        IF (GetNeedNetReduction (TypeId.Key, false),
          ORDER (SetNeedNetReduction (TypeId.Key, false, false),
                 PTGReductionTplDef(TypeId.Ptg, PTGStr ("Net"))),
          PTGNULL))
      DEPENDS_ON ReductionFunctionDef.coded;
   ReductionFunctionDef.code = PTGSeq (.smallcode, .largecode);
 END;

 RULE rReductionFunctionBody :
   ReductionFunctionBody ::=  Statements
 COMPUTE
   ReductionFunctionBody.parcode = Statements.parcode;
 END;
 }
```

This macro is defined in definitions 285, 286, 287, 289, 291, and 293.
This macro is invoked in definition 334.

46.4 Winner-takes-all functions

The structure of the code generation for winner-takes-all functions corresponds directly to that for reduction functions.

290 *Subroutine Definition PTG*[290] ≡

```
{
 WtaFunctionDef:  [IndentNewLine] "plural Bool p_" $2
            " (plural " $1 "* ME, plural " $1 "* YOU)"
            [IndentNewLine] "{" [IndentIncr]
            $3 [IndentDecr]
            [IndentNewLine] "}" [IndentNewLine]
 WtaTplDef: [IndentNewLine] [IndentNewLine] "#define _type_ " $1
            [IndentNewLine] "#include \"Wta" $2 ".tpl\""
            [IndentNewLine] [IndentNewLine]
 }
```

This macro is defined in definitions 282, 283, 284, 288, 290, and 292.
This macro is invoked in definition 336.

Subroutine Definition Generation[291] ≡ 291

```
{
 RULE rWtaFunctionDef :
   WtaFunctionDef ::=  TypeId 'WTA' NewWtaFunctionId 'IS'
                       WtaFunctionBody 'END' OptWTA
 COMPUTE
   .smallcode = /* the wta function itself */
      PTGWtaFunctionDef (TypeId.Ptg, NewWtaFunctionId.Ptg,
                         WtaFunctionBody.parcode);
   .largecode = /* the a_WTA function(s) (if needed) */
      PTGSeq3 (
        IF (GetNeedConWta (TypeId.Key, false),
          ORDER (SetNeedConWta (TypeId.Key, false, false),
                 PTGWtaTplDef(TypeId.Ptg, PTGStr ("Con"))),
          PTGNULL),
        IF (GetNeedNodeWta (TypeId.Key, false),
          ORDER (SetNeedNodeWta (TypeId.Key, false, false),
                 PTGWtaTplDef(TypeId.Ptg, PTGStr ("Node"))),
          PTGNULL),
        IF (GetNeedNetWta (TypeId.Key, false),
          ORDER (SetNeedNetWta (TypeId.Key, false, false),
                 PTGWtaTplDef(TypeId.Ptg, PTGStr ("Net"))),
          PTGNULL))
      DEPENDS_ON WtaFunctionDef.coded;
   WtaFunctionDef.code = PTGSeq (.smallcode, .largecode);
 END;

 RULE rWtaFunctionBody :
   WtaFunctionBody ::=  Statements
 COMPUTE
   WtaFunctionBody.parcode = Statements.parcode;
 END;
 }
```

This macro is defined in definitions 285, 286, 287, 289, 291, and 293.
This macro is invoked in definition 334.

46.5 Merge procedures

For **MERGE** procedures, we have to implement a normal and a virtualized version (much like for other object procedures). The virtualized version is implemented with a template. If no merge procedure is defined for a connection, node, or network type, we generate a dummy version (plus the full virtualized version) in the rule of the type definition body; we cannot save the virtualized version, because we also use it to help construct the replicates in the network replication operation.

Subroutine Definition PTG[292] ≡ 292

```
{
 MergeProcDef:
   [IndentNewLine] "plural void MERGE_" $1 "(plural " $1 " *ME, "
                   "plural " $1 " *YOU)"
   [IndentNewLine] "{" [IndentIncr]
                   $2/*body*/ [IndentDecr]
   [IndentNewLine] "}"
```

```
    [IndentNewLine] [IndentNewLine]
  }
```

This macro is defined in definitions 282, 283, 284, 288, 290, and 292.
This macro is invoked in definition 336.

293 *Subroutine Definition Generation*[293] ≡

```
  {
  RULE rMergeProcDef :
    MergeProcDef ::=  'MERGE' 'IS' MergeProcedureBody 'END' OptMERGE
  COMPUTE
    MergeProcDef.code =
      PTGMergeProcDef (INCLUDING TypeDefBody.InhPtg, MergeProcedureBody.parcode);
  END;

  RULE rMergeProcedureBody :
    MergeProcedureBody ::=  Statements
  COMPUTE
    MergeProcedureBody.parcode = Statements.parcode;
  END;
  }
```

This macro is defined in definitions 285, 286, 287, 289, 291, and 293.
This macro is invoked in definition 334.

47 Statements

In statement code generation we always have the distinction between code generated for a sequential context (attribute `seqcode`) and code generated for a parallel context (attribute `parcode`).

294 *Statement PTG*[294] ≡

{
Assignment PTG[296]
I/O Assignment PTG[298]
Procedure Call PTG[300]
Reduction Statement PTG[304]
Winner-Takes-All Statement PTG[306]
Control Flow PTG[308]
Data Allocation Statement PTG[312]
Merge Statement PTG[314]
}

This macro is defined in definitions 294.
This macro is invoked in definition 336.

295 *Statement Generation*[295] ≡

```
  {
  RULE rStatements :
    Statements ::=  DataObjectDefList StatementList
  COMPUTE
    Statements.seqcode =
      PTGSeq (DataObjectDefList CONSTITUENTS DataObjectDef.seqcode SHIELD ()
              WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull),
              CONSTITUENTS (Assignment.seqcode,
                InputAssignment.seqcode, OutputAssignment.seqcode,
                ProcedureCall.seqcode, ObjectProcedureCall.seqcode,
                ReductionStmt.seqcode, WtaStmt.seqcode,
                ReturnStmt.seqcode, IfStmt.seqcode,
```

```
              LoopStmt.seqcode, BreakStmt.seqcode,
              DataAllocationStmt.seqcode, MergeStmt.seqcode)
            WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull));
  Statements.parcode =
    PTGSeq (DataObjectDefList CONSTITUENTS DataObjectDef.parcode SHIELD ()
            WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull),
            CONSTITUENTS (Assignment.parcode,
              InputAssignment.parcode, OutputAssignment.parcode,
              ProcedureCall.parcode, ObjectProcedureCall.parcode,
              ReductionStmt.parcode, WtaStmt.parcode,
              ReturnStmt.parcode, IfStmt.parcode,
              LoopStmt.parcode, BreakStmt.parcode,
              DataAllocationStmt.parcode, MergeStmt.parcode)
            WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull));
END;
```

Assignment Generation[297]
I/O Assignment Generation[299]
Procedure Call Generation[301]
Reduction Statement Generation[305]
Winner-Takes-All Statement Generation[307]
Control Flow Generation[309]
Data Allocation Statement Generation[313]
Merge Statement Generation[315]
}

This macro is invoked in definition 334.

47.1 Assignment

All assignment operators are implemented in a straightforward manner. Note, though, that the `%=` assignment does not work for floats in MPL.

Assignment PTG[296] ≡ 296

{

```
Assign:   [IndentNewLine] $1 $2 $3 ";"
```

}

This macro is invoked in definition 294.

Assignment Generation[297] ≡ 297

{

```
RULE rAssignment :
  Assignment ::=  Object AssignOperator Expr
COMPUTE
  Assignment.seqcode =
     PTGAssign (Object.seqcode, AssignOperator.code, Expr.seqcode);
  Assignment.parcode =
     PTGAssign (Object.parcode, AssignOperator.code, Expr.parcode);
END;

RULE rAssignOp:       AssignOperator ::=  ':=' COMPUTE
  AssignOperator.code = PTGStr (" = ");
END;

RULE rPlusAssignOp:   AssignOperator ::=  '+=' COMPUTE
  AssignOperator.code = PTGStr (" += ");
```

```
END;

RULE rMinusAssignOp:  AssignOperator ::=  '-=' COMPUTE
  AssignOperator.code = PTGStr (" -= ");
END;

RULE rMulAssignOp:    AssignOperator ::=  '*=' COMPUTE
  AssignOperator.code = PTGStr (" *= ");
END;

RULE rDivAssignOp:    AssignOperator ::=  '/=' COMPUTE
  AssignOperator.code = PTGStr (" /= ");
END;

RULE rModAssignOp:    AssignOperator ::=  '%=' COMPUTE
  AssignOperator.code = PTGStr (" %= ");  /* doesn't work for float !!! */
END;
}
```

This macro is invoked in definition 295.

47.2 I/O assignment

The I/O assignment operators are implemented as a call to the corresponding input or output procedure.

298 *I/O Assignment PTG*[298] ≡

```
{
IOAssign:
    [IndentNewLine] $/*IN/OUT*/ $/*type*/ "(&" $/*group_D*/ "_D, "
                    "(plural char*)_sgl(" $/*base*/ "), " [IndentIncr]
    [IndentNewLine] $/*slice*/ ", offsetof(" $/*nd_type*/ "," $/*field*/ "),"
    [IndentNewLine] "offsetof ( " $/*nd_type*/ ",_me_D), "
                    "sizeof(" $/*nd_type*/ "), " $/*IOvar*/ ");" [IndentDecr]
}
```

This macro is invoked in definition 294.

299 *I/O Assignment Generation*[299] ≡

```
{
RULE rInputAssignment :
  InputAssignment ::=  Object '<--' Object
COMPUTE
  InputAssignment.seqcode =
    PTGIOAssign (PTGStr ("INPUT_"), PTGKey (Object[2].Type),
                 Object[1].largecode,
                 Object[1].largecode, Object[1].seqslicecode,
                 PTGKey (Object[1].ParVarType), Object[1].smallcode,
                 PTGKey (Object[1].ParVarType), PTGKey (Object[1].ParVarType),
                 Object[2].seqcode);
  InputAssignment.parcode = PTGNULL; /* cannot occur */
END;

RULE rOutputAssignment :
  OutputAssignment ::=  Object '-->' Object
COMPUTE
  OutputAssignment.seqcode =
    PTGIOAssign (PTGStr ("OUTPUT_"), PTGKey (Object[2].Type),
```

```
                    Object[1].largecode,
                    Object[1].largecode, Object[1].seqslicecode,
                    PTGKey (Object[1].ParVarType), Object[1].smallcode,
                    PTGKey (Object[1].ParVarType), PTGKey (Object[1].ParVarType),
                    Object[2].seqcode);
    OutputAssignment.parcode = PTGNULL; /* cannot occur */
  END;
  }
```

This macro is invoked in definition 295.

47.3 Procedure call

Procedure calls to global procedures are translated one to one.

Procedure Call PTG[300] ≡ 300

```
{
ProcedureCall: [IndentNewLine] $1 " (" $2 ");"
}
```

This macro is defined in definitions 300 and 302.
This macro is invoked in definition 294.

Procedure Call Generation[301] ≡ 301

```
{
RULE rProcedureCall :
  ProcedureCall ::=  ProcedureId '(' ArgumentList ')'
COMPUTE
  ProcedureCall.seqcode =
    PTGProcedureCall (ProcedureId.Ptg, ArgumentList.seqcode);
  ProcedureCall.parcode =
    PTGProcedureCall (PTGParName (ProcedureId.Ptg), ArgumentList.parcode);
END;
}
```

This macro is defined in definitions 301 and 303.
This macro is invoked in definition 295.

Object procedure calls have to be treated differently not only depending on whether we call from sequential or parallel context. We must also discriminate whether we call within the same level of parallelism (e.g. a node procedure from within a node procedure) or to a level that extends the parallelism (e.g. a connection procedure from within a node procedure or a network procedure from sequential context). Another discrimination to be made is that between local and remote connection procedure calls.

Procedure Call PTG[302] ≡ 302

```
{
RecProcedureCall:
   [IndentNewLine] $1 "_" $2 " (" $3 [IndentIncr] $4 [IndentDecr] ");"
pRecProcedureCall:
   [IndentNewLine] "p_" $1 "_" $2 " (" $3 [IndentIncr] $4 [IndentDecr] ");"
ConProcedureCall:
   [IndentNewLine] $1 "_" $2 " (&" $3 [IndentIncr] $4 [IndentDecr] ");"
aConProcedureCall:
   [IndentNewLine] "a_" $1 "_" $2 " (_sgl(" $3 "), _sgl(" $4 "_D.con_ls)"
   [IndentIncr]    $5 [IndentDecr] ");"
arConProcedureCall:
   [IndentNewLine] "a_" $1 "_remote_" $2 " (_sgl(" $3 "), _sgl(" $4 "_D.con_ls)"
   [IndentIncr]    $5 [IndentDecr] ");"
NodeProcedureCall:
```

```
    [IndentNewLine] $1 "_" $2 " (&" $3 [IndentIncr] $4 [IndentDecr] ");"
  aNodeProcedureCall:
    [IndentNewLine] "a_" $1 "_" $2 " (_sgl(" $3 ")," [IndentIncr]
    [IndentNewLine] "_sgl(" $3 "_D.localsizeN), " $4 $5 [IndentDecr] ");"
  NetProcedureCall:
    [IndentNewLine] $1 "_" $2 " (&" $3 [IndentIncr] $4 [IndentDecr] ");"
  aNetProcedureCall:
    [IndentNewLine] "if (_IntervalInOp (" $3 "._me_D.meI, " $4 "))" [IndentIncr]
    [IndentNewLine] $1 "_" $2 " (&" $3 [IndentIncr]
                    $5 [IndentDecr] [IndentDecr] ");"
  }
```

This macro is defined in definitions 300 and 302.
This macro is invoked in definition 294.

303 *Procedure Call Generation*[303] ≡

```
{
  RULE rObjectProcedureCall :
    ObjectProcedureCall ::=  Object '.' ObjectProcedureId '(' ArgumentList ')'
  COMPUTE
    .code = PTGKey (Object.Type) DEPENDS_ON Object.known;
    .Dataloc = GetDataloc (Object.Type, NoMode) /* relevant for con only */
               DEPENDS_ON INCLUDING CupitProgram.allknown;
    ObjectProcedureCall.seqcode =
      IF (EQ (.Kind, RecordTypeK),
        PTGRecProcedureCall (ObjectProcedureId.Ptg, .code, Object.seqcode,
                             WithComma (ArgumentList.seqcode)),
      /* else */
      IF (EQ (.Kind, ConTypeK),
        PTGStr ("/* ERROR: sequential call of connection procedure !!! */"),
      /* else */
      IF (EQ (.Kind, NodeTypeK),
        PTGStr ("/* ERROR: sequential call of node procedure !!! */"),
      /* else */
      IF (EQ (.Kind, NetTypeK),
        PTGaNetProcedureCall (ObjectProcedureId.Ptg, .code, Object.seqcode,
                              Object.parslicecode,
                              WithComma (ArgumentList.seqcode)),
      /* else */
        ORDER (
          Message1 (DEADLY, "Object kind %d at seqObjectProcedureCall", .Kind),
          PTGNULL)))));
    ObjectProcedureCall.parcode =
      IF (EQ (.Kind, RecordTypeK),
        PTGpRecProcedureCall (ObjectProcedureId.Ptg, .code, Object.parcode,
                              WithComma (ArgumentList.parcode)),
      /* else */
      IF (EQ (.Kind, ConTypeK),
        IF (EQ (Object.Kind, ParVariableK),
          IF (EQ (Object.Mode, .Dataloc),
            PTGaConProcedureCall (ObjectProcedureId.Ptg, .code, Object.parcode,
                                  Object.smallcode,
                                  WithComma (ArgumentList.parcode)),
            PTGarConProcedureCall (ObjectProcedureId.Ptg, .code, Object.parcode,
                                   Object.smallcode,
                                   WithComma(ArgumentList.parcode))),
          PTGConProcedureCall (ObjectProcedureId.Ptg, .code, Object.parcode,
```

```
                              WithComma (ArgumentList.parcode))),
        /* else */
        IF (EQ (.Kind, NodeTypeK),
          IF (EQ (Object.Kind, ParVariableK),
            PTGaNodeProcedureCall (ObjectProcedureId.Ptg, .code, Object.smallcode,
                                   Object.parslicecode,
                                   WithComma (ArgumentList.parcode)),
            PTGNodeProcedureCall (ObjectProcedureId.Ptg, .code, Object.parcode,
                                  WithComma (ArgumentList.parcode))),
        /* else */
        IF (EQ (.Kind, NetTypeK),
          PTGNetProcedureCall (ObjectProcedureId.Ptg, .code, Object.parcode,
                               WithComma (ArgumentList.parcode)),
        /* else */
          ORDER (
            Message1 (DEADLY, "Object kind %d at parObjectProcedureCall", .Kind),
            PTGNULL)))));
    END;
    }
```

This macro is defined in definitions 301 and 303.
This macro is invoked in definition 295.

No code generation for `MultiObjProcedureCall`s is necessary, because they are treated in exactly the same way as a sequence of calls on the SIMD machine. We make no attempts whatsoever to parallelize this stuff (not even if it is just several calls to different slices of the same node group). The sequentialization of `MultiObjProcedureCall`s is covered by the CONSTITUENTS construct at `statements` above.

47.4 Reduction statement

Reductions from sequential context are possible for networks only. Reductions from parallel context may be for nodes or for connections. Reductions on connections may be local or remote. However, we must not generate an error message for other cases (e.g. for a sequential connection reduction) because code for these cases can legitimately be produced in this rule here as long as it is not used later! Therefore, the code generated in these cases is just a comment saying what case it is.

The number of arguments for the PTG functions is so large for the reduction statements that we prefer to repeat some of them but be allowed to give them in the same order in which they appear in the generated code.

Reduction Statement PTG[304] ≡ 304

```
  {
   ConReductionStmt:
      [IndentNewLine] "p_REDUCTION_" $/*rtype*/ "_connections (" [IndentIncr]
      [IndentNewLine] "&ME->_me_D, &" $/*interf_D*/ "_D, "
                      "(plural char*)_sgl(" $/*base*/ "),"
      [IndentNewLine] "offsetof(" $/*type*/ "," $/*field*/ "), "
                      "offsetof(" $/*type*/ ",_me_D.exists),"
      [IndentNewLine] "sizeof(" $/*type*/ "), false/*is_remote*/,"
      [IndentNewLine] "p_" $/*op*/ ", &" $/*into*/ ");" [IndentDecr]
   rConReductionStmt:
      [IndentNewLine] "p_REDUCTION_" $/*rtyp*/ "_connections (" [IndentIncr]
      [IndentNewLine] "&ME->_me_D, &" $/*interf_D*/ "_D, "
                      "(plural char*)_sgl(" $/*base*/ "),"
      [IndentNewLine] "offsetof(" $/*type*/ "," $/*field*/ "), "
                      "offsetof(_remote_connection,_me_D.exists),"
      [IndentNewLine] "sizeof(_remote_connection), true/*is_remote*/,"
```

```
    [IndentNewLine] "p_" $/*op*/ ",& " $/*into*/ ");" [IndentDecr]
 NodeReductionStmt:
    [IndentNewLine] "p_REDUCTION_" $/*rtype*/ "_nodes (" [IndentIncr]
    [IndentNewLine] "&" $/*group_D*/ "_D, " $/*slice*/ ", "
                    "(plural char*)_sgl(" $/*base*/ "), "
    [IndentNewLine] "offsetof(" $/*type*/ "," $/*field*/ "), "
                    "offsetof(" $/*type*/ ",_me_D),"
    [IndentNewLine] "sizeof(" $/*type*/ "),"
    [IndentNewLine] "p_" $/*op*/ ", &" $/*into*/ ");" [IndentDecr]
 NetReductionStmt:
    [IndentNewLine] "p_REDUCTION_" $/*rtype*/ "_networks (" [IndentIncr]
    [IndentNewLine] "&" $/*net*/ "._me_D, " $/*slice*/ ", "
                    "&" $/*net*/ "." $/*field*/ ","
    [IndentNewLine] "p_" $/*op*/ ", &" $/*into*/ ");" [IndentDecr]
 }
```

This macro is invoked in definition 294.

305 *Reduction Statement Generation*[305] ≡

```
{
 RULE rReductionStmt :
   ReductionStmt ::=  'REDUCTION' Object ':' ReductionFunctionId 'INTO' Object
 COMPUTE
   .Dataloc = GetDataloc (Object[1].ParVarType, NoMode)
              DEPENDS_ON INCLUDING CupitProgram.allknown;
   .Ptg = PTGKey (Object[1].ParVarType);
   ReductionStmt.seqcode =
     IF (EQ (.Kind, ConTypeK),
       PTGStr ("/* ERROR: sequential call of connection reduction !!! */"),
     /* else */
     IF (EQ (.Kind, NodeTypeK),
       PTGStr ("/* ERROR: sequential call of node reduction !!! */"),
     /* else */
     IF (EQ (.Kind, NetTypeK),
       PTGNetReductionStmt (PTGKey (Object[2].Type),
           Object[1].largecode, Object[1].seqslicecode,
           Object[1].largecode, Object[1].smallcode,
           ReductionFunctionId.Ptg, Object[2].seqcode),
     /* else */
       ORDER (
         Message1 (FATAL, "Object kind %d at seqReductionStmt", .Kind),
         PTGNULL))));
   ReductionStmt.parcode =
     IF (EQ (.Kind, ConTypeK),
       IF (EQ (Object[1].ParVarMode, .Dataloc),
         PTGConReductionStmt (PTGKey (Object[2].Type),
             Object[1].largecode, Object[1].largecode,
             .Ptg, Object[1].smallcode, .Ptg, .Ptg,
             ReductionFunctionId.Ptg, Object[2].seqcode),
         PTGrConReductionStmt (PTGKey (Object[2].Type),
             Object[1].largecode, Object[1].largecode,
             .Ptg, Object[1].smallcode,
             ReductionFunctionId.Ptg, Object[2].seqcode)),
     /* else */
     IF (EQ (.Kind, NodeTypeK),
       PTGNodeReductionStmt (PTGKey (Object[2].Type),
           Object[1].largecode, Object[1].parslicecode,
```

```
          Object[1].largecode, .Ptg, Object[1].smallcode, .Ptg, .Ptg,
          ReductionFunctionId.Ptg, Object[2].seqcode),
    /* else */
    IF (EQ (.Kind, NetTypeK),
      PTGStr ("/* ERROR: parallel call of network reduction !!! */"),
    /* else */
      ORDER (
        Message1 (FATAL, "Object kind %d at parReductionStmt", .Kind),
        PTGNULL))));
END;
}
```

This macro is invoked in definition 295.

47.5 Winner-takes-all statement

The remarks given above for reduction statements also apply for winner-takes-all statements. In addition, we have to act according to the result of the WTA function call: On those PEs where the WTA call returned a non-zero result, the generated code has to call the object procedure given in the original WTA statement.

Winner-Takes-All Statement PTG[306] ≡ 306

```
{
 ConWtaStmt:
 /* Procedure call: copy _w_ into local object _x_ for the call and either
    restore it afterwards or set local and remote 'exists' marker to false.
    Copying is necessary to have a singular object pointer to work on. !!!
    You might use the a_* procedure to achieve remote exists marker setting.
 */
    [IndentNewLine] "{ plural void* plural _w_;"
    [IndentNewLine] " _w_=p_WTA_" $/*rtype*/ "_connections ("
    [IndentIncr] [IndentIncr]
    [IndentNewLine] "&ME->_me_D, &" $/*interf_D*/ "_D, "
                    "(plural char*)_sgl(" $/*base*/ "),"
    [IndentNewLine] "offsetof(" $/*type*/ "," $/*field*/ "), "
                    "offsetof(" $/*type*/ ",_me_D.exists),"
    [IndentNewLine] "sizeof(" $/*type*/ "), false/*is_remote*/, "
                    "p_" $/*op*/ ");" [IndentDecr] [IndentDecr]
    [IndentNewLine] " if (_w_ != 0)" [IndentIncr]
    [IndentNewLine] $/*proc*/ "_" $/*type*/
                    "((plural " $/*type*/ "* plural)_w_, "
    [IndentIncr]    $/*args*/ [IndentDecr] ");" [IndentDecr]
    [IndentNewLine] "}"
 rConWtaStmt:
 /* see above !!! */
    [IndentNewLine] "{ plural void* plural _w_;"
    [IndentNewLine] " _w_=p_WTA_" $/*rtype*/ "_connections ("
    [IndentIncr] [IndentIncr]
    [IndentNewLine] "&ME->_me_D, &" $/*interf_D*/ "_D, "
                    "(plural char*)_sgl(" $/*base*/ "),"
    [IndentNewLine] "offsetof(" $/*type*/ "," $/*field*/ "), "
                    "offsetof(_remote_connection,_me_D.exists),"
    [IndentNewLine] "sizeof(_remote_connection), true/*is_remote*/, "
                    "p_" $/*op*/ ");" [IndentDecr] [IndentDecr]
    [IndentNewLine] " if (_w_ != 0)" [IndentIncr]
    [IndentNewLine] "a_" $/*proc*/ "_remote_" $/*type*/
```

```
                    "((plural _remote_connection* plural)_w_, 1, "
   [IndentIncr]     $/*args*/ [IndentDecr] ");" [IndentDecr]
   [IndentNewLine] "}"
NodeWtaStmt:
/* procedure call cannot be performed directly because object pointer must be
   singular. We therefore perform a loop similar to that used in the
   a_* procedure. !!!  How handle REPLICATE 0 within ???
*/
   [IndentNewLine] "{ plural void* plural _w_;"
   [IndentNewLine] " _w_=p_WTA_" $/*rtype*/ "_nodes ("
   [IndentIncr] [IndentIncr]
   [IndentNewLine] $/*group_D*/ "_D, " $/*slice*/ ", "
                    "(plural char*)_sgl(" $/*base*/ "),"
   [IndentNewLine] "offsetof(" $/*type*/ "," $/*field*/ "), "
                    "offsetof(" $/*type*/ ",_me_D),"
   [IndentNewLine] "sizeof(" $/*type*/ "),"
   [IndentNewLine] "p_" $/*op*/ ");" [IndentDecr] [IndentDecr]
   [IndentNewLine] " if (_w_ != 0)" [IndentIncr]
   [IndentNewLine] $/*proc*/ "_" $/*type*/
                    "((plural " $/*type*/ "* plural)_w_, "
   [IndentIncr]     $/*args*/ [IndentDecr] ");" [IndentDecr]
   [IndentNewLine] "}"
NetWtaStmt:
   [IndentNewLine] "{ plural _bool _w_;"
   [IndentNewLine] " _w_=p_WTA_" $/*rtype*/ "_networks ("
   [IndentIncr] [IndentIncr]
   [IndentNewLine] $/*net*/ "._me_D, " $/*slice*/ ", "
                    "(plural char*)&" $/*net*/ "." $/*field*/ ","
   [IndentNewLine] "p_" $/*op*/ ");" [IndentDecr] [IndentDecr]
   [IndentNewLine] " if (_w_)" [IndentIncr]
   [IndentNewLine] $/*proc*/ "_" $/*type*/ "(&" $/*net*/ ", "
   [IndentIncr]     $/*args*/ [IndentDecr] ");" [IndentDecr]
   [IndentNewLine] "}"
}
```

This macro is invoked in definition 294.

307 *Winner-Takes-All Statement Generation*[307] ≡

```
{
 RULE rWtaStmt :
   WtaStmt ::=  'WTA' Object ':' Elementname '.' WtaFunctionId ':'
                ObjectProcedureId '(' ArgumentList ')'
 COMPUTE
   .Dataloc = GetDataloc (Object[1].ParVarType, NoMode)
              DEPENDS_ON INCLUDING CupitProgram.allknown;
   .Ptg = PTGKey (Object[1].ParVarType);
   WtaStmt.seqcode =
     IF (EQ (.Kind, ConTypeK),
       PTGStr ("/* ERROR: sequential call of connection WTA !!! */"),
     /* else */
     IF (EQ (.Kind, NodeTypeK),
       PTGStr ("/* ERROR: sequential call of node WTA !!! */"),
     /* else */
     IF (EQ (.Kind, NetTypeK),
       PTGNetWtaStmt (PTGKey (Elementname.Type),
           Object[1].smallcode, Object[1].seqslicecode,
           Object[1].smallcode, Elementname.Ptg,
```

```
            WtaFunctionId.Ptg, ObjectProcedureId.Ptg, .Ptg,
            Object[1].smallcode, ArgumentList.parcode),
      /* else */
        ORDER (
          Message1 (FATAL, "Object kind %d at seqWtaStmt", .Kind),
          PTGNULL))));
    WtaStmt.parcode =
      IF (EQ (.Kind, ConTypeK),
        IF (EQ (Object[1].ParVarMode, .Dataloc),
          PTGConWtaStmt (PTGKey (Elementname.Type),
              Object[1].parcode, Object[1].parcode,
              .Ptg, Elementname.Ptg, .Ptg, .Ptg,
              WtaFunctionId.Ptg, ObjectProcedureId.Ptg, .Ptg, .Ptg,
              ArgumentList.parcode),
          PTGrConWtaStmt (PTGKey (Elementname.Type),
              Object[1].parcode, Object[1].parcode, .Ptg, Elementname.Ptg,
              WtaFunctionId.Ptg, ObjectProcedureId.Ptg, .Ptg,
              ArgumentList.parcode)),
      /* else */
      IF (EQ (.Kind, NodeTypeK),
        PTGNodeWtaStmt (PTGKey (Elementname.Type),
            Object[1].smallcode, Object[1].parslicecode,
            Object[1].smallcode, .Ptg, Elementname.Ptg, .Ptg, .Ptg,
            WtaFunctionId.Ptg, ObjectProcedureId.Ptg, .Ptg, .Ptg,
            ArgumentList.parcode),
      /* else */
      IF (EQ (.Kind, NetTypeK),
        PTGStr ("/* ERROR: parallel call of network WTA !!! */"),
      /* else */
        ORDER (
          Message1 (FATAL, "Object kind %d at parWtaStmt", .Kind),
          PTGNULL))));
  END;
  }
```

This macro is invoked in definition 295.

47.6 Control flow

The translation of control flow statements is more or less straightforward. As usual, we must discriminate a sequential and a parallel version. All loops are implemented as MPL `while` loops. If the actual `WHILE` part is missing in the CuPit program, we must nevertheless generate a *plural* condition in the dummy `while` of the parallel version (or otherwise no `BREAK` in a plural `if` while be allowed).

The translation of the `FOR` loop is done using the equivalent code pattern given in the language definition. Since this is a little more complicated, it is discussed separately below.

Control Flow PTG[308] ≡ 308

```
{
 Return0:         [IndentNewLine] "return;"
 Return:          [IndentNewLine] "return (" $1 ");"
 IfStmt:          [IndentNewLine] "if (" $1 ") {" [IndentIncr]
                  $2 [IndentDecr] [IndentNewLine] "}" $3
 ElsifPart:       [IndentNewLine] "else if (" $1 ") {" [IndentIncr]
                  $2 [IndentDecr] [IndentNewLine] "}" $3
 ElsePart:        [IndentNewLine] "else {" [IndentIncr]
```

```
                 $1 [IndentDecr] [IndentNewLine] "}"
WhileLoop:       $1 "{" [IndentIncr] $2 [IndentDecr] [IndentNewLine] "}"
While:           [IndentNewLine] "while (" $1 ") "
WhileOS:         [IndentNewLine] "while (1) "
WhileOP:         [IndentNewLine] "while ((plural int) 1) "
Until:           [IndentNewLine] "/* UNTIL: */ if (" $1 ")  break;"
Break:           [IndentNewLine] "break;"
}
```

This macro is defined in definitions 308 and 310.
This macro is invoked in definition 294.

309 *Control Flow Generation*[309] ≡

```
{
 RULE rReturnStmtVoid :
   ReturnStmt ::=  'RETURN'
 COMPUTE
   ReturnStmt.seqcode = PTGReturn0 ();
   ReturnStmt.parcode = PTGReturn0 ();
 END;

 RULE rReturnStmt :
   ReturnStmt ::=  'RETURN' Expr
 COMPUTE
   ReturnStmt.seqcode = PTGReturn (Expr.seqcode);
   ReturnStmt.parcode = PTGReturn (Expr.parcode);
 END;

 RULE rIfStmt :
   IfStmt ::=  'IF' Expr 'THEN' Statements ElsePart 'END' OptIF
 COMPUTE
   IfStmt.seqcode = PTGIfStmt (Expr.seqcode, Statements.seqcode,
                               ElsePart.seqcode);
   IfStmt.parcode = PTGIfStmt (Expr.parcode, Statements.parcode,
                               ElsePart.parcode);
 END;

 RULE rElsif :
   ElsePart ::=  'ELSIF' Expr 'THEN' Statements ElsePart
 COMPUTE
   ElsePart[1].seqcode =
      PTGElsifPart (Expr.seqcode, Statements.seqcode, ElsePart[2].seqcode);
   ElsePart[1].parcode =
      PTGElsifPart (Expr.parcode, Statements.parcode, ElsePart[2].parcode);
 END;

 RULE rElse0 :
   ElsePart ::=
 COMPUTE
   ElsePart.seqcode = PTGNULL;
   ElsePart.parcode = PTGNULL;
 END;

 RULE rElse :
   ElsePart ::=  'ELSE' Statements
 COMPUTE
   ElsePart.seqcode = PTGElsePart (Statements.seqcode);
```

```
  ElsePart.parcode = PTGElsePart (Statements.parcode);
END;

RULE rLoopStmt :
  LoopStmt ::=  OptWhilePart 'REPEAT' Statements OptUntilPart 'END' OptREPEAT
COMPUTE
  LoopStmt.seqcode = PTGWhileLoop (OptWhilePart.seqcode,
                         PTGSeq (Statements.seqcode, OptUntilPart.seqcode));
  LoopStmt.parcode = PTGWhileLoop (OptWhilePart.parcode,
                         PTGSeq (Statements.parcode, OptUntilPart.parcode));
END;

RULE rOptWhilePart :
  OptWhilePart ::=  'WHILE' Expr
COMPUTE
  OptWhilePart.seqcode = PTGWhile (Expr.seqcode);
  OptWhilePart.parcode = PTGWhile (Expr.parcode);
END;

RULE rOptWhilePart0 :
  OptWhilePart ::=
COMPUTE
  OptWhilePart.seqcode = PTGWhile0S ();
  OptWhilePart.parcode = PTGWhile0P ();
END;

RULE rOptUntilPart :
  OptUntilPart ::=  'UNTIL' Expr
COMPUTE
  OptUntilPart.seqcode = PTGUntil (Expr.seqcode);
  OptUntilPart.parcode = PTGUntil (Expr.parcode);
END;

RULE rOptUntilPart0 :
  OptUntilPart ::=
COMPUTE
  OptUntilPart.seqcode = PTGNULL;
  OptUntilPart.parcode = PTGNULL;
END;

RULE rBreakStmt :
  BreakStmt ::=  'BREAK'
COMPUTE
  BreakStmt.seqcode = PTGBreak ();
  BreakStmt.parcode = PTGBreak ();
END;
}
```

This macro is defined in definitions 309 and 311.
This macro is invoked in definition 295.

To implement the translation pattern for `FOR` loops as given in the language definition, we need an additional variable of the same type as the count variable. The name of this variable is created in the `ForLoopStep` symbol (simply because it only appears in the `FOR` loop context) by inheriting from the symbols `GenName` and `IdPtg` from the Eli tech specification module library. To allow this variable declaration in the target code, we have to surround the whole code segment by braces. The `code` and `smallcode` symbols in `ForLoopStep` are used to deliver the value to be added to the loop count variable and the test to apply to loop count variable and loop limit, respectively.

310 *Control Flow PTG*[310] ≡

```
{
 /* parameters: $1:loop variable, $2:startexpression, $3:step, $4:endexpression,
    $5:statements, $6:until-test, $7:comparison-operator, $8:endconstant,
    $9:loopvariabletype, $10:plural-or-not
 */
 ForLoop:  [IndentNewLine] "{ " $10 $9 "  " $8 "; /* FOR-limit container */"
           [IndentNewLine] $1 " = " $2 "; /* initialization */"
           [IndentNewLine] $8 " = " $4 "; /* limit computation */"
           [IndentNewLine] "while (" $1 $7 $8 ") {  /* FOR termination test */"
           [IndentIncr]    $5  "  /* statements */"
           $6  /* UNTIL test, if any */
           [IndentNewLine] $1 " += " $3 ";  /* count step */"
           [IndentDecr] [IndentNewLine] "}}  /* End of FOR */"
}
```

This macro is defined in definitions 308 and 310.
This macro is invoked in definition 294.

311 *Control Flow Generation*[311] ≡

```
{
 RULE rForLoopStmt :
   LoopStmt ::=  'FOR' Object ':=' Expr ForLoopStep Expr 'REPEAT'
                 Statements OptUntilPart 'END' OptREPEAT
 COMPUTE
   LoopStmt.seqcode =
     PTGForLoop (Object.seqcode, Expr[1].seqcode, ForLoopStep.code,
                 Expr[2].seqcode, Statements.seqcode, OptUntilPart.seqcode,
                 ForLoopStep.smallcode, ForLoopStep.Ptg, PTGKey (Object.Type),
                 PTGStr (""));
   LoopStmt.parcode =
     PTGForLoop (Object.parcode, Expr[1].parcode, ForLoopStep.code,
                 Expr[2].parcode, Statements.parcode, OptUntilPart.parcode,
                 ForLoopStep.smallcode, ForLoopStep.Ptg, PTGKey (Object.Type),
                 PTGStr ("plural "));
 END;

 SYMBOL ForLoopStep INHERITS GenName, IdPtg END;

 RULE rUpto :   ForLoopStep ::=  'UPTO'
 COMPUTE
   ForLoopStep.code = PTGStr ("1");
   ForLoopStep.smallcode = PTGStr (" <= ");
 END;

 RULE rTo :     ForLoopStep ::=  'TO'
 COMPUTE
   ForLoopStep.code = PTGStr ("1");
   ForLoopStep.smallcode = PTGStr (" <= ");
 END;

 RULE rDownto : ForLoopStep ::=  'DOWNTO'
 COMPUTE
   ForLoopStep.code = PTGStr ("-1");
   ForLoopStep.smallcode = PTGStr (" >= ");
```

```
END;
}
```

This macro is defined in definitions 309 and 311.
This macro is invoked in definition 295.

47.7 Data allocation

The connection replicate and disconnect data allocation statements are compiled into calls to the run time system functions for the respective operations. The other data allocation statements are implemented as calls to procedures that are individually defined for the respective type.

Data Allocation Statement PTG[312] ≡ 312

```
{
 ReplicateConStmt:
    [IndentNewLine] "REPLICATE_connection (&ME->_me_D, " $1 ");"
 ReplicateNodeStmt:
    [IndentNewLine] "REPLICATE_" $1 " (ME, " $2 ");"
 ReplicateNetStmt:
    [IndentNewLine] "REPLICATE_" $ " (&" $ ", " $ ", true);"
 ReplicateNetInterval2Stmt:
    [IndentNewLine] "REPLICATE_" $ " (&" $ ", _CanonicInterval2 (" $ "), true);"
 ReplicateNetInterval1Stmt:
    [IndentNewLine] "REPLICATE_" $ " (&" $ ", _CanonicInterval1 (" $ "), true);"
 ReplicateNetIntStmt:
    [IndentNewLine] "REPLICATE_" $1 " (&" $2
                    ", _CanonicIntervalInt ((Int)(" $3 ")), " $3 " != 1);"
 ExtendStmt:
    [IndentNewLine] "EXTEND_" $1 " (&ME->_me_D, &" $2 ", &" $2 "_D, " $3 ");"
 ConnectStmt:
    /* Procedure parameters: contype, network_D,
        nodes1, ndsize1, slice1, group_D1, nd_D_off1, interf_D_off1, interf_off1,
        nodes2, ndsize2, slice2, group_D2, nd_D_off2, interf_D_off2, interf_off2
       Template parameters: 1:contype,
        2:node_group1, 3:node_type1, 4:slice1, 5:interf1,
        6:node_group2, 7:node_type2, 8:slice2, 9:interf2
    */
    [IndentNewLine] "CONNECT_" $1 " (&ME->_me_D," [IndentIncr]
    [IndentNewLine] "(plural char*)_sgl(" $2 "), sizeof(" $3 "), "
                    $4 ", &" $2 "_D, "
    [IndentNewLine] "offsetof(" $3 ",_me_D), offsetof (" $3 "," $5 "_D), "
                    "offsetof (" $3 "," $5 "), "
    [IndentNewLine] "(plural char*)_sgl(" $6 "), sizeof(" $7 "), "
                    $8 ", &" $6 "_D, "
    [IndentNewLine] "offsetof(" $7 ",_me_D), offsetof (" $7 "," $9 "_D), "
                    "offsetof (" $7 "," $9 "));" [IndentDecr]
 DisconnectStmt:
    /* Procedure parameters: contype, network_D,
        nodes1, slice1, group_D1, nd_D_off1, interf_D_off1, interf_off1,
        nodes2, slice2, group_D2, nd_D_off2, interf_D_off2, interf_off2,
        consize1, oe_offset1, exists_offset1,
        consize2, exists_offset2
       Template parameters: 1:contype,
        2:node_group1, 3:node_type1, 4:slice1, 5:interf1,
        6:node_group2, 7:node_type2, 8:slice2, 9:interf2
    */
```

```
    [IndentNewLine] "DISCONNECT (&ME->_me_D," [IndentIncr]
    [IndentNewLine] "(plural char*)_sgl(" $2 "), "
                    $4 ", &" $2 "_D, "
    [IndentNewLine] "offsetof(" $3 ",_me_D), offsetof (" $3 "," $5 "_D), "
                    "offsetof (" $3 "," $5 "), "
    [IndentNewLine] "(plural char*)_sgl(" $6 "), "
                    $8 ", &" $6 "_D, "
    [IndentNewLine] "offsetof(" $7 ",_me_D), offsetof (" $7 "," $9 "_D), "
                    "offsetof (" $7 "," $9 "), "
    [IndentNewLine] "sizeof(" $1 "), offsetof(" $1 ",_oe), "
                    "offsetof(" $1 ",_me_D.exists)"
    [IndentNewLine] "sizeof(_remote_connection), "
                    "offsetof(_remote_connection,_me_D.exists));" [IndentDecr]
}
```

This macro is invoked in definition 294.

313 *Data Allocation Statement Generation*[313] ≡

```
{
 RULE rReplicateInto :
   DataAllocationStmt ::=  'REPLICATE' Object 'INTO' Expr
 COMPUTE
   DataAllocationStmt.seqcode =
     IF (EQ (.Kind, ConTypeK),
       PTGStr ("/* ERROR: sequential call of connection replication */"),
     /* else */
     IF (EQ (.Kind, NodeTypeK),
       PTGStr ("/* ERROR: sequential call of node replication */"),
     /* else */
     IF (EQ (.Kind, NetTypeK),
       IF (EQ (Expr.Type, IntervalKey),
         PTGReplicateNetStmt (PTGKey (Object.Type), Object.seqcode,
                              Expr.seqcode),
       IF (EQ (Expr.Type, Interval2Key),
         PTGReplicateNetInterval2Stmt (PTGKey (Object.Type), Object.seqcode,
                                       Expr.seqcode),
       IF (EQ (Expr.Type, Interval1Key),
         PTGReplicateNetInterval1Stmt (PTGKey (Object.Type), Object.seqcode,
                                       Expr.seqcode),
       IF (IsInt (Expr.Type),
         PTGReplicateNetIntStmt (PTGKey (Object.Type), Object.seqcode,
                                 Expr.seqcode),
         /* else */
         ORDER (
           Message (DEADLY, "Impossible Expr.Type at REPLICATE"),
           PTGNULL))))),
     /* else */
       ORDER (
         Message1 (DEADLY, "Object kind %d at seqReplicateStmt", .Kind),
         PTGNULL))));
   DataAllocationStmt.parcode =
     IF (EQ (.Kind, ConTypeK),
       PTGReplicateConStmt (Expr.parcode),
     /* else */
     IF (EQ (.Kind, NodeTypeK),
       PTGReplicateNodeStmt (PTGKey (Object.Type), Expr.parcode),
     /* else */
```

```
    IF (EQ (.Kind, NetTypeK),
      PTGStr ("/* ERROR: parallel call of network replication */"),
    /* else */
      ORDER (
        Message1 (DEADLY, "Object kind %d at parReplicateStmt", .Kind),
        PTGNULL))));
END;

RULE rExtendBy :
  DataAllocationStmt ::=  'EXTEND' Object 'BY' Expr
COMPUTE
  DataAllocationStmt.seqcode =
    PTGStr ("/* ERROR: sequential call of EXTEND */");
  DataAllocationStmt.parcode =
    PTGExtendStmt (PTGKey (GetType (Object.Type, NoKey)),
                   Object.parcode, Expr.parcode)
    DEPENDS_ON INCLUDING CupitProgram.allknown;
END;

RULE rConnectTo :
  DataAllocationStmt ::=  'CONNECT' Object 'TO' Object
COMPUTE
  DataAllocationStmt.seqcode =
    PTGStr ("/* ERROR: sequential call of CONNECT */");
  DataAllocationStmt.parcode =
    IF (EQ (GetDataloc (Object[1].Type, NoMode), OutMode),
      PTGConnectStmt (PTGKey (Object[1].Type),
        Object[1].largecode, PTGKey (Object[1].ParVarType),
        Object[1].parslicecode, Object[1].smallcode,
        Object[2].largecode, PTGKey (Object[2].ParVarType),
        Object[2].parslicecode, Object[2].smallcode),
      PTGConnectStmt (PTGKey (Object[1].Type),
        Object[2].largecode, PTGKey (Object[2].ParVarType),
        Object[2].parslicecode, Object[2].smallcode,
        Object[1].largecode, PTGKey (Object[1].ParVarType),
        Object[1].parslicecode, Object[1].smallcode))
      DEPENDS_ON INCLUDING CupitProgram.allknown;
END;

RULE rDisconnectFrom :
  DataAllocationStmt ::=  'DISCONNECT' Object 'FROM' Object
COMPUTE
  DataAllocationStmt.seqcode =
    PTGStr ("/* ERROR: sequential call of DISCONNECT */");
  DataAllocationStmt.parcode =
    IF (EQ (GetDataloc (Object[1].Type, NoMode), OutMode),
      PTGDisconnectStmt (PTGKey (Object[1].Type),
        Object[1].largecode, PTGKey (Object[1].ParVarType),
        Object[1].parslicecode, Object[1].smallcode,
        Object[2].largecode, PTGKey (Object[2].ParVarType),
        Object[2].parslicecode, Object[2].smallcode),
      PTGDisconnectStmt (PTGKey (Object[1].Type),
        Object[2].largecode, PTGKey (Object[2].ParVarType),
        Object[2].parslicecode, Object[2].smallcode,
        Object[1].largecode, PTGKey (Object[1].ParVarType),
```

```
        Object[1].parslicecode, Object[1].smallcode))
      DEPENDS_ON INCLUDING CupitProgram.allknown;
END;
}
```

This macro is invoked in definition 295.

47.8 Merge statement

The MERGE statement is simply translated into a call to the a_MERGE procedure defined for the respective network type.

314 *Merge Statement PTG*[314] ≡

```
{
MergeStmt:
  [IndentNewLine] "a_MERGE_" $1 " (&" $2 ", true, true, false);"
}
```

This macro is invoked in definition 294.

315 *Merge Statement Generation*[315] ≡

```
{
RULE rMergeStmt:
  MergeStmt ::=  'MERGE' Object
COMPUTE
  MergeStmt.seqcode = PTGMergeStmt (PTGKey (Object.Type), Object.seqcode);
  MergeStmt.parcode = PTGStr ("/* ERROR: parallel call of MERGE */");
END;
}
```

This macro is invoked in definition 295.

48 Expressions

316 *Expression PTG*[316] ≡

```
{
VarArg:        "&(" $1 ")"
}
```

This macro is defined in definitions 316, 321, and 323.
This macro is invoked in definition 336.

317 *Expression Generation*[317] ≡

```
{
RULE rObjectExpr :
  Expr ::=  Object
COMPUTE
  Expr.seqcode = Object.seqcode;
  Expr.parcode = Object.parcode;
END;

RULE rDenoterExpr :
  Expr ::=  Denoter
COMPUTE
  Expr.seqcode = Denoter.seqcode;
  Expr.parcode = Denoter.parcode;
END;
```

```
RULE rIntDenoter :
  Denoter ::=  IntegerDenoter
COMPUTE
  Denoter.seqcode = IntegerDenoter.Ptg;
  Denoter.parcode = Denoter.seqcode;
END;

RULE rRealDenoter :
  Denoter ::=  RealDenoter
COMPUTE
  Denoter.seqcode = RealDenoter.Ptg;
  Denoter.parcode = Denoter.seqcode;
END;

RULE rStringDenoter :
  Denoter ::=  StringDenoter
COMPUTE
  Denoter.seqcode = StringDenoter.Ptg;
  Denoter.parcode = Denoter.seqcode;  /* cast necessary !!!? */
END;
}
```

This macro is defined in definitions 317, 318, 322, and 324.
This macro is invoked in definition 334.

In argument passing, we use the address of arguments to `VAR` and `IO` parameters and the object itself otherwise:

Expression Generation[318] ≡ 318

```
{
RULE rActParam :
  ActParam IS Expr
COMPUTE
  ActParam.seqcode =
    IF (OR (IsVarAcc (.Access), IsIoAcc (.Access)),
      PTGVarArg (Expr.seqcode),
      Expr.seqcode);
  ActParam.parcode =
    IF (OR (IsVarAcc (.Access), IsIoAcc (.Access)),
      PTGVarArg (Expr.parcode),
      Expr.parcode);
END;

RULE rArgumentList :
  ArgumentList ::= ExprList
COMPUTE
  TRANSFER seqcode, parcode;
END;

RULE rArgumentList0 :
  ArgumentList ::=
COMPUTE
  ArgumentList.seqcode = PTGNULL;
  ArgumentList.parcode = PTGNULL;
END;

RULE rExprList1 :
```

```
  ExprList ::=  ActParam
COMPUTE
  TRANSFER seqcode, parcode;
END;

RULE rExprList :
  ExprList ::=  ExprList ',' ActParam
COMPUTE
  ExprList[1].seqcode = PTGList (ExprList[2].seqcode, ActParam.seqcode);
  ExprList[1].parcode = PTGList (ExprList[2].parcode, ActParam.parcode);
END;
}
```

This macro is defined in definitions 317, 318, 322, and 324.
This macro is invoked in definition 334.

48.1 Operators

The code for expressions with operators is generated in two different forms: Those CuPit operators that have a direct MPL equivalent are turned into exactly analog MPL code. The others are implemented as function or macro calls (see the header file of the run time system `rts.h`). The textual representations of the operators and the distinction between builtin representations are implemented in the following module in the two functions `OpRepresentation` and `IsBuiltin`.

319 **operatorgen.h**[319] ≡

```
{
#ifndef operatorgen_H
#define operatorgen_H

#include "cupit.h"
#include "oiladt2.h"

Bool   IsBuiltin (char *operator_representation);
char*  OpRepresentation (tOilOp op, tOilOp indication);
#endif
}
```

This macro is attached to an output file.

320 **operatorgen.c**[320] ≡

```
{
#include <stdio.h>
#include "operatorgen.h"
#include "OilDecls.h"

Bool IsBuiltin (char *operator_representation)
{
  /* idea: all non-builtin operator names begin with an underscore */
  _assert (operator_representation != 0 && *operator_representation != 0);
  return (*operator_representation != '_');
}


char* OpRepresentation (tOilOp op, tOilOp indication)
{
  int nr = OilOpName(op),
      ind;
```

```
switch (nr) {
  case bAndOp_name:     return ("&&");
  case bOrOp_name:      return ("||");
  case bXorOp_name:     return ("!=");
  case iEqOp_name:
  case rEqOp_name:      return ("==");
  case iNeqOp_name:
  case rNeqOp_name:     return ("!=");
  case IntervalEqOp_name:
  case RealervalEqOp_name:      return ("_IntervalEqOp");
  case IntervalNeqOp_name:
  case RealervalNeqOp_name:     return ("_IntervalNeqOp");
  case StringEqOp_name:         return ("_StringEqOp");
  case iLtOp_name:
  case rLtOp_name:      return ("<");
  case iGtOp_name:
  case rGtOp_name:      return (">");
  case iLeOp_name:
  case rLeOp_name:      return ("<=");
  case iGeOp_name:
  case rGeOp_name:      return (">=");
  case IntInOp1_name:
  case IntInOp2_name:
  case IntInOp_name:
  case RealInOp_name:   return ("_IntervalInOp");
  case IntIntervalOp1_name:     return ("_IntIntervalOp1");
  case IntIntervalOp2_name:     return ("_IntIntervalOp2");
  case IntIntervalOp_name:      return ("_IntIntervalOp");
  case RealIntervalOp_name:     return ("_RealIntervalOp");
  case iBitandOp_name:  return ("&");
  case iBitorOp_name:   return ("|");
  case iBitxorOp_name:  return ("^");
  case iLshiftOp_name:  return ("<<");
  case iRshiftOp_name:  return (">>");
  case iPlusOp_name:
  case rPlusOp_name:    return ("+");
  case iMinusOp_name:
  case rMinusOp_name:   return ("-");
  case iMulOp_name:
  case rMulOp_name:     return ("*");
  case iDivOp_name:
  case rDivOp_name:     return ("/");
  case iModOp_name:     return ("%");
  case rModOp_name:     return ("_realmod");
  case iExpOp_name:     return ("_iExpOp");
  case riExpOp_name:    return ("_riExpOp");
  case rExpOp_name:     return ("_rExpOp");
  case iNegOp_name:
  case rNegOp_name:     return ("-");
  case bNotOp_name:     return ("!");
  case iBitnotOp_name:  return ("~");
  case iMinOp_name:
  case iMinOp1_name:
  case iMinOp2_name:
  case rMinOp_name:     return ("_MinOp");
```

```
    case iMaxOp_name:
    case iMaxOp1_name:
    case iMaxOp2_name:
    case rMaxOp_name:     return ("_MaxOp");
    case iRandomOp_name:  return ("_iRandomOp");
    case iRandomOp1_name: return ("_iRandomOp1");
    case iRandomOp2_name: return ("_iRandomOp2");
    case rRandomOp_name:  return ("_rRandomOp");
    default:
      ind = OilOpName(indication);
      if (ind == EqOp_name)  /* for symbolic types */
        return ("==");
      else if (ind == NeqOp_name)
        return ("!=");
      else {
        fprintf (stderr, "Operator number: %d   ", nr);
         _assert (false); return ("UnknownOperator");
      }
  }
}
}
```

This macro is attached to an output file.

The actual code generation templates contain one template each for the ternary operator (**PTGTernOp**), binary operators that are builtin into MPL (**PTGBinOpBuiltin**), binary operators that are implemented via function or macro calls (**PTGBinOpCall**) and again the same two for unary operators (**PTGUnOpBuiltin** and **PTGUnOpCall**)

321 *Expression PTG*[321] ≡

```
{
 TernOp:        "(" $1 " ? " $2 " : " $3 ")"
 BinOpBuiltin:  "(" $1 " " $2 " " $3 ")"
 BinOpCall:     $1 "(" $2 "," $3 ")"
 UnOpBuiltin:   "(" $1 $2 ")"
 UnOpCall:      $1 "(" $2 ")"
 TypeCast:      "_mk" $1 "(" $2 ")"
 pTypeCast:     "p_mk" $1 "(" $2 ")"
 NetMaxindex:   "_sgl(" $1 "._me_D.repN-1)"
 ArrayMaxindex: "(" $1 "-1)"
 GroupMaxindex: "((plural int)" $1 "_D.nodesN-1)"
 GroupMaxindexSeq: "_sgl((plural int)" $1 "_D.nodesN-1)"
 InterfaceMaxindex:  "(update_conI_conN((plural char*)_sgl(" $1 "),"
                       "sizeof(" $2 "), offsetof(" $2 ",_me_D),"
                       "&" $1 "_D), (plural int)" $1 "_D.conN-1)"
}
```

This macro is defined in definitions 316, 321, and 323.
This macro is invoked in definition 336.

The results of constant folding are used only for the non-interval types, because for integer interval types we would have to perform additional type discrimination.

322 *Expression Generation*[322] ≡

```
{
 RULE rTernaryExpr :
   Expr ::=  Expr '?' Expr ':' Expr
 COMPUTE
   Expr[1].seqcode =
```

```
    IF (NOT (OR (IsErrorConst (Expr[1].Val), IsInterval (Expr[1].Type))),
      PTGStr (Const2Str (Expr[1].Val)),
    /* else */
      PTGTernOp (Expr[2].seqcode, Expr[3].seqcode, Expr[4].seqcode));
  Expr[1].parcode =
    IF (NOT (OR (IsErrorConst (Expr[1].Val), IsInterval (Expr[1].Type))),
      PTGStr (Const2Str (Expr[1].Val)),
    /* else */
      PTGTernOp (Expr[2].parcode, Expr[3].parcode, Expr[4].parcode));
END;

RULE rBinaryExpr :
  Expr ::=  Expr BinOp Expr
COMPUTE
  .str = OpRepresentation (Expr[1].Operator, BinOp.Operator);
  .code = PTGStr (.str);
  Expr[1].seqcode =
    IF (NOT (OR (IsErrorConst (Expr[1].Val), IsInterval (Expr[1].Type))),
      PTGStr (Const2Str (Expr[1].Val)),
    /* else */
    IF (IsBuiltin (.str),
      PTGBinOpBuiltin (Expr[2].seqcode, .code, Expr[3].seqcode),
      PTGBinOpCall (.code, Expr[2].seqcode, Expr[3].seqcode)));
  Expr[1].parcode =
    IF (NOT (OR (IsErrorConst (Expr[1].Val), IsInterval (Expr[1].Type))),
      PTGStr (Const2Str (Expr[1].Val)),
    /* else */
    IF (IsBuiltin (.str),
      PTGBinOpBuiltin (Expr[2].parcode, .code, Expr[3].parcode),
      PTGBinOpCall (PTGSeq (PTGStr("p"), .code),
                    Expr[2].parcode, Expr[3].parcode)));
END;

RULE rUnaryExpr :
  Expr ::=  UnaryOp Expr
COMPUTE
  .str = OpRepresentation (Expr[1].Operator, UnaryOp.Operator);
  .code = PTGStr (.str);
  Expr[1].seqcode =
    IF (NOT (OR (IsErrorConst (Expr[1].Val), IsInterval (Expr[1].Type))),
      PTGStr (Const2Str (Expr[1].Val)),
    /* else */
    IF (IsBuiltin (.str),
      PTGUnOpBuiltin (.code, Expr[2].seqcode),
      PTGUnOpCall (.code, Expr[2].seqcode)));
  Expr[1].parcode =
    IF (NOT (OR (IsErrorConst (Expr[1].Val), IsInterval (Expr[1].Type))),
      PTGStr (Const2Str (Expr[1].Val)),
    /* else */
    IF (IsBuiltin (.str),
      PTGUnOpBuiltin (.code, Expr[2].parcode),
      PTGUnOpCall (PTGSeq (PTGStr("p"), .code), Expr[2].parcode)));
END;

RULE rTypeConvExpr :
```

```
  Expr ::=  TypeId '(' ExprList ')'
COMPUTE
  Expr[1].seqcode = PTGTypeCast (TypeId.Ptg, ExprList.seqcode);
  Expr[1].parcode = PTGpTypeCast (TypeId.Ptg, ExprList.parcode);
END;

RULE rMaxindexExpr :
  Expr ::=  'MAXINDEX' '(' Object ')'
COMPUTE
  Expr[1].seqcode =
    IF (OR (EQ (.Kind, NodeArrayTypeK), EQ (.Kind, NodeGroupTypeK)),
      PTGGroupMaxindexSeq (Object.seqcode),
    /* else */
    IF (EQ (.Kind, ConTypeK),
      PTGStr ("/* ERROR: sequential call of MAXINDEX (ConTypeK) !!! */"),
    /* else */
    IF (EQ (.Kind, ArrayTypeK),
      PTGArrayMaxindex (PTGInt (GetIval (GetVal (Object.Type, ErrorConst)))),
    /* else */
    IF (EQ (.Kind, NetTypeK),
      PTGNetMaxindex (Object.seqcode),
    /* else */
      ORDER (
        Message1 (DEADLY, "Object kind %d at seqMAXINDEX", .Kind),
        PTGNULL)))));
  Expr[1].parcode =
    IF (OR (EQ (.Kind, NodeArrayTypeK), EQ (.Kind, NodeGroupTypeK)),
      PTGGroupMaxindex (Object.parcode),
    /* else */
    IF (EQ (.Kind, ConTypeK),
      PTGInterfaceMaxindex (Object.parcode,
         IF (EQ (Object.Mode, GetDataloc (Object.Type, NoMode)),
           PTGKey (Object.Type), PTGStr ("_remote_connection"))),
    /* else */
    IF (EQ (.Kind, ArrayTypeK),
      PTGArrayMaxindex (PTGInt (GetIval (GetVal (Object.Type, ErrorConst)))),
    /* else */
    IF (EQ (.Kind, NetTypeK),
      PTGNetMaxindex (Object.seqcode),
    /* else */
      ORDER (
        Message1 (DEADLY, "Object kind %d at seqMAXINDEX", .Kind),
        PTGNULL)))));
END;
}
```

This macro is defined in definitions 317, 318, 322, and 324.
This macro is invoked in definition 334.

48.2 Function call

323 *Expression PTG*[323] $\equiv$

```
{
FunctionCall:        [IndentNewLine] $1 " (" $2 ")"
ObjectFunctionCall:  [IndentNewLine] $1 "_" $2 " (" $3 [IndentIncr]
                                     $4 [IndentDecr] ")"
```

}

This macro is defined in definitions 316, 321, and 323.
This macro is invoked in definition 336.

Expression Generation[324] ≡ 324

```
{
RULE rFCallExpr :
  Expr ::=  FunctionCall
COMPUTE
  Expr[1].seqcode = FunctionCall.seqcode;
  Expr[1].parcode = FunctionCall.parcode;
END;

RULE rObjFCallExpr:
  Expr ::=  ObjectFunctionCall
COMPUTE
  Expr[1].seqcode = ObjectFunctionCall.seqcode;
  Expr[1].parcode = ObjectFunctionCall.parcode;
END;

RULE rFunctionCall :
  FunctionCall ::=  FunctionId '(' ArgumentList ')'
COMPUTE
  FunctionCall.seqcode =
    PTGFunctionCall (FunctionId.Ptg, ArgumentList.seqcode);
  FunctionCall.parcode =
    PTGFunctionCall (PTGParName (FunctionId.Ptg), ArgumentList.parcode);
END;

RULE rObjectFunctionCall1 :
  ObjectFunctionCall ::=  Object '.' ObjectFunctionId '(' ArgumentList ')'
COMPUTE
  ObjectFunctionCall.seqcode =
    PTGObjectFunctionCall (ObjectFunctionId.Ptg, PTGKey (Object.Type),
                           Object.seqcode, WithComma (ArgumentList.seqcode));
  ObjectFunctionCall.parcode =
    PTGObjectFunctionCall (ObjectFunctionId.Ptg, PTGKey (Object.Type),
                           Object.parcode, WithComma (ArgumentList.parcode));
END;
}
```

This macro is defined in definitions 317, 318, 322, and 324.
This macro is invoked in definition 334.

49 Objects

Object Generation[325] ≡ 325

```
{
SYMBOL Object COMPUTE  /* default values: */
  THIS.smallcode = PTGNULL;
  THIS.largecode = PTGNULL;
  THIS.seqslicecode = PTGStr ("_allslice");
  THIS.parslicecode = PTGStr ("p__allslice");
END;
}
```

This macro is defined in definitions 325, 327, 329, 331, and 332.
This macro is invoked in definition 334.

49.1 ME, YOU, INDEX, and explicit variables

The objects `ME` and `YOU` are always passed as pointers. That means that to access their fields, we must either use the arrow syntax `ME->` of MPL or must first dereference the pointer `(*ME)`. We use the latter approach, which, although less elegant, is significantly simpler.

326 *Object PTG*[326] ≡

```
{
PtrObject:   "*(" $1 ")"
}
```

This macro is defined in definitions 326, 328, and 330.
This macro is invoked in definition 336.

327 *Object Generation*[327] ≡

```
{
RULE rMeObject :
  Object ::=  'ME'
COMPUTE
  Object.seqcode = PTGStr ("(*ME)");
  Object.parcode = PTGStr ("(*ME)");
END;

RULE rYouObject :
  Object ::=  'YOU'
COMPUTE
  Object.seqcode = PTGStr ("(*YOU)");
  Object.parcode = PTGStr ("(*YOU)");
END;

RULE rIndexObject :
  Object ::=  'INDEX'
COMPUTE
  Object.seqcode = PTGStr ("ME->_me_D.meI");
  Object.parcode = PTGStr ("ME->_me_D.meI");
END;

RULE rDirectObject :
  Object ::=  Objectname
COMPUTE
  Object.seqcode =
    IF (OR (EQ (Objectname.Access, VarPAcc), EQ (Objectname.Access, IoPAcc)),
      PTGPtrObject (Objectname.Ptg),
    /* else */
    IF (NOT (OR (IsErrorConst (Objectname.Val), IsInterval (Objectname.Type))),
      PTGStr (Const2Str (Objectname.Val)),
    /* else */
      Objectname.Ptg));
  Object.parcode = Object.seqcode;
END;
}
```

This macro is defined in definitions 325, 327, 329, 331, and 332.
This macro is invoked in definition 334.

49.2 Selection

For the selection there are three cases: (1) Ordinary selection on a local record, (2) selection to form parallel variable selections, and (3) selection from an explicit network variable in sequential context. The first case is what the code generated here represents, the second case is handled by making the individual parts of the selection expression available in `seqslicecode` or `parslicecode` and in `smallcode` and `largecode`. These attributes are then used later to generate the code that uses the parallel variable selection.

Object PTG[328] ≡ 328

```
{
 Selection:      $1 "." $2
 NetSelection:   $1 "." $2  /* how do this properly??? proc[0] fails
                               if $1 is subscribed */
}
```

This macro is defined in definitions 326, 328, and 330.
This macro is invoked in definition 336.

Object Generation[329] ≡ 329

```
{
 RULE rSelectionObject :
   Object ::=  Object '.' Elementname
 COMPUTE
   Object[1].seqslicecode = Object[2].seqslicecode;
   Object[1].parslicecode = Object[2].parslicecode;
   Object[1].seqcode =
      IF (EQ (.Kind, NetTypeK),
      PTGNetSelection (Object[2].seqcode, Elementname.Ptg),
      PTGSelection (Object[2].seqcode, Elementname.Ptg));
   Object[1].parcode =
      PTGSelection (Object[2].parcode, Elementname.Ptg);
   Object[1].smallcode = Elementname.Ptg;
   Object[1].largecode = Object[2].smallcode;
 END;
}
```

This macro is defined in definitions 325, 327, 329, 331, and 332.
This macro is invoked in definition 334.

49.3 Subscription

For the subscription there are two cases: Ordinary subscription on a local array and subscription to form parallel variables. The first case is what the code generated here represents, the second case is handled by making the individual parts of the subscription expression available in `seqslicecode` or `parslicecode` and in `smallcode`. These attributes are used when it later turns out that the subscription has to be implemented as a parallel variable in an object procedure call or in a parallel variable selection. The interval given is converted to the type Interval if it is an Interval1 or Interval2.

Object PTG[330] ≡ 330

```
{
 CanonicalizeInterval:  $3 "_Canonic" $2 "(" $1 ")"
 Subscription:          $1 "[" $2 "]"
 NetSubscription:       "proc[_pick_PE(" $1 "._me_D.meI==" $2 ")]." $1
}
```

This macro is defined in definitions 326, 328, and 330.
This macro is invoked in definition 336.

Object Generation[331] ≡ 331

```
{
 RULE rIndexedObject :
   Object ::=  Object '[' Expr ']'
 COMPUTE
   Object[1].seqcode =
     IF (EQ (.Kind, NetTypeK),
       PTGNetSubscription (Object[2].seqcode, Expr.seqcode),
       PTGSubscription (Object[2].seqcode, Expr.seqcode));
   Object[1].parcode = PTGSubscription (Object[2].parcode, Expr.parcode);
   Object[1].seqslicecode =
     IF (EQ (Expr.Type, Interval1Key),
       PTGCanonicalizeInterval (Expr.seqcode, PTGKey (Expr.Type), PTGStr("")),
     /* else */
     IF (EQ (Expr.Type, Interval2Key),
       PTGCanonicalizeInterval (Expr.seqcode, PTGKey (Expr.Type), PTGStr("")),
     /* else */
       Expr.seqcode));
   Object[1].parslicecode =
     IF (EQ (Expr.Type, Interval1Key),
       PTGCanonicalizeInterval (Expr.parcode, PTGKey (Expr.Type), PTGStr("p")),
     /* else */
     IF (EQ (Expr.Type, Interval2Key),
       PTGCanonicalizeInterval (Expr.parcode, PTGKey (Expr.Type), PTGStr("p")),
     /* else */
       Expr.parcode));
   Object[1].smallcode    = Object[2].parcode;
 END;

 RULE rUnindexedObject :
   Object ::=  Object '[' ']'
 COMPUTE
   TRANSFER seqcode, parcode;
   Object[1].smallcode = Object[2].parcode;
 END;
}
```

This macro is defined in definitions 325, 327, 329, 331, and 332.
This macro is invoked in definition 334.

49.4 Connection addressing

The direct connection addressing is not implemented.

332 *Object Generation*[332] ≡

```
{
 RULE rWeightObject :
   Object ::=  '{' Object '-->' Object '}'
 COMPUTE
   Object[1].seqcode = PTGSeq (Object[2].seqcode, PTGStr ("~")); /* Dummy !!! */
   Object[1].parcode = PTGSeq (Object[2].parcode, PTGStr ("~")); /* Dummy !!! */
 END;
}
```

This macro is defined in definitions 325, 327, 329, 331, and 332.
This macro is invoked in definition 334.

50 Basic syntactic elements

This section describes the code generation for the non-constant basic tokens, i.e., for denoters (literals) and identifiers. The `IdPtg` symbol is from the tech library and generates a PTG node that contains the string represented by the `Sym` attribute value of the same symbol. `NumPtg` converts the actual value of the `Sym` attribute into a decimal digit string to represent an integer and `CStringPtg` generates a C style string literal.

Basic Token Generation[333] ≡ 333

```
{
 SYMBOL NewTypeId        INHERITS IdPtg END;
 SYMBOL NewEnumId        INHERITS IdPtg END;
 SYMBOL NewElemId        INHERITS IdPtg END;
 SYMBOL NewInterfaceId   INHERITS IdPtg END;
 SYMBOL NewDataId        INHERITS IdPtg END;
 SYMBOL NewParamId       INHERITS IdPtg END;
 SYMBOL NewProcedureId   INHERITS IdPtg END;
 SYMBOL NewFunctionId    INHERITS IdPtg END;
 SYMBOL NewObjProcedureId      INHERITS IdPtg END;
 SYMBOL NewObjFunctionId       INHERITS IdPtg END;
 SYMBOL NewReductionFunctionId INHERITS IdPtg END;
 SYMBOL NewWtaFunctionId INHERITS IdPtg END;
 SYMBOL TypeId           INHERITS IdPtg END;
 SYMBOL Objectname       INHERITS IdPtg END;
 SYMBOL Elementname      INHERITS IdPtg END;
 SYMBOL ProcedureId      INHERITS IdPtg END;
 SYMBOL ObjectProcedureId      INHERITS IdPtg END;
 SYMBOL FunctionId       INHERITS IdPtg END;
 SYMBOL ObjectFunctionId       INHERITS IdPtg END;
 SYMBOL ReductionFunctionId    INHERITS IdPtg END;
 SYMBOL WtaFunctionId    INHERITS IdPtg END;

 SYMBOL RealDenoter      INHERITS IdPtg END;
 SYMBOL IntegerDenoter   INHERITS NumPtg END;
 SYMBOL StringDenoter    INHERITS CStringPtg END;
}
```

This macro is invoked in definition 334.

51 Put it all together

All of the above specifications belong into a small number of files, to which they are now assigned.

The LIDO file contains the code generation attribution rules.

code2.lido[334] ≡ 334

{
Code Generation Attributes[249]
Coding Order[254]
Overall Program Generation[256]
Type Definition Generation[261]
Data Object Definition Generation[278]
Subroutine Definition Generation[285]
Statement Generation[295]
Expression Generation[317]

Object Generation[325]
Basic Token Generation[333]
}

This macro is attached to an output file.

The head file is included at the top of the C code that is generated from the lido file.

335 **code2.head**[335] ≡
{
```
#include "clp.h"
#include "divgen.h"
#include "operatorgen.h"
#include "remoteconcomm.h"
```
alignment computation[251]
}

This macro is attached to an output file.

The PTG file contains the code generation templates.

336 **code2.ptg**[336] ≡
{
Cupit Program PTG[255]
Type Definition PTG[260]
Data Object Definition PTG[277]
Subroutine Definition PTG[282]
Statement PTG[294]
Expression PTG[316]
Object PTG[326]
}

This macro is attached to an output file.

The property definition language PDL file contains the definition table entries introduced by the code generation.

337 **code2.pdl**[337] ≡
{
```
"ptg_gen.h"  /* PTGNode */
```
Code Generation Properties[250]
}

This macro is attached to an output file.

The C file (plus corresponding `.h` file) contains the procedure that sets the size and alignment property for the predefined types.

338 **code2.h**[338] ≡
{
```
#ifndef type_H
#define type_H
extern void SetPredefTypeSizeAlign ();
#endif
```
}

This macro is attached to an output file.

339 **code2.c**[339] ≡
{
```
#include "scope.h"    /* Declarations of predefined objects' xKey variables */
#include "pdl_gen.h"  /* Property manipulation functions and property types */
```

Set Properties of Predefined Types[252]
}

This macro is attached to an output file.

PART V: Run Time System

The run time system consists of functions that are used by the generated code but are not directly included in it. Instead, the functions are split across a number files which are compiled only once (and independent of a run of the CuPit compiler) and are then linked to the code generated by the CuPit compiler when the CuPit compiler is used.

In this part we also find the Makefiles used for several administrative tasks and the compiler driver shell script.

52 Language operations

This section describes that part of the run time system that implements operations that are directly visible in CuPit source programs. These are the random number generation (including a language-invisible initialization function), the interval operations, type conversions for the standard types, some operators, and topology change operations (including some language-invisible auxiliary operations).

52.1 Random number generation

340 *rts.h Random Number Generator*[340] ≡

```
{
 Real             _rRandomOp (Realerval range);
 plural Real      p_rRandomOp (plural Realerval range);
 Int              _iRandomOp (Interval range);
 plural Int       p_iRandomOp (plural Interval range);
 Int1             _iRandomOp1 (Interval1 range);
 plural Int1      p_iRandomOp1 (plural Interval1 range);
 Int2             _iRandomOp2 (Interval2 range);
 plural Int2      p_iRandomOp2 (plural Interval2 range);
 void             _INITRANDOM(Int seed);
}
```

This macro is invoked in definition 385.

There is one singular and one plural function for each of the interval types plus the initialization operation. The initialization is called once in the global initialization function of the CuPit program and can be called again in an external function by the user if necessary. The initialization uses a seed supplied by the caller for reproducable results or the time of day if the seed 0 is given. The random Reals have five random digits. Note that this random number generator is not sophisticated, as its properties are usually not very important for neural algorithms.

341 *RTS Random Number Generator*[341] ≡

```
{
 extern long random ();
 extern srandom();
 visible extern int time (/* int* */);

 #define _Realrandomprecision  99999.0

 Real _rRandomOp (Realerval range)
 {
   return (‘random01 * (range.max-range.min) + range.min);

 ‘random01:
   /* a random factor in the range [0,1) : */
```

```
  (Real)_iRandomOp (_IntIntervalOp (0, _Realrandomprecision))/
  _Realrandomprecision
}

plural Real p_rRandomOp (plural Realerval range)
{
  return (`random01 * (range.max-range.min) + range.min);

`random01:
  /* a random factor in the range [0,1) : */
  (plural Real)p_iRandomOp (p_IntIntervalOp (0, _Realrandomprecision)) /
  _Realrandomprecision
}

Int _iRandomOp (Interval range)
{
  return (((Int)random() % (range.max-range.min+1)) + range.min);
}

plural Int p_iRandomOp (plural Interval range)
{
  return (((plural Int)p_random() % (range.max-range.min+1)) + range.min);
}

Int1 _iRandomOp1 (Interval1 range)
{
  return (((Int)random() % (range.max-range.min+1)) + range.min);
}

plural Int1 p_iRandomOp1 (plural Interval1 range)
{
  return (((plural Int)p_random() % (range.max-range.min+1)) + range.min);
}

Int2 _iRandomOp2 (Interval2 range)
{
  return (((Int)random() % (range.max-range.min+1)) + range.min);
}

plural Int2 p_iRandomOp2 (plural Interval2 range)
{
  return (((plural Int)p_random() % (range.max-range.min+1)) + range.min);
}

void _INITRANDOM(Int seed)
{
  /* if seed == 0, the front end time is used as init value */
  plural int  r = 0;
  plural Bool new = false; /* where has a new seed just been put ? */
  int step;
  srandom (seed ? (int)seed
               : (int)callRequest (time, sizeof(int*), (int*)0));
  /* now seed p_random on all processors */
  proc[0].r = random ();
  proc[0].new = true;
```

```
  if (iyproc == 0) {  /* in first row only: */
   for (step = 1; step < nxproc; step <<= 1) {
      if (ixproc < step) {
        if (new) {
          p_srandom (r);
          new = false;
        }
        xnetE[step].r = p_random();
        xnetE[step].new = true;
      }
    }
  }
  for (step = 1; step < nyproc; step <<= 1) {
    if (iyproc < step) {
      if (new) {
        p_srandom (r);
        new = false;
      }
      xnetS[step].r = p_random();
      xnetS[step].new = true;
    }
  }
  if (new)
    p_srandom (r);
}
}
```

This macro is invoked in definition 386.

52.2 Type conversion and operators

All conversion operations for the standard types are implemented as macros.

342 *rts.h Type Conversions*[342] ≡

```
{
#define _mkReal(x) ((Real)(x))
#define _mkInt(x) ((Int)(x))
#define _mkInt2(x) ((Int2)(x))
#define _mkInt1(x) ((Int1)(x))
#define p_mkReal(x) ((plural Real)(x))
#define p_mkInt(x) ((plural Int)(x))
#define p_mkInt2(x) ((plural Int2)(x))
#define p_mkInt1(x) ((plural Int1)(x))
}
```

This macro is invoked in definition 385.

Some of the operators are implemented as macros, too, since they are available in MPL or can easily be implemented as a single expression. Others are implemented as functions. Note that some of the macro implementations can induce unexpected side effects if used carelessly, because the operands are evaluated more than once.

343 *rts.h Operators*[343] ≡

```
{
#define _IntervalEqOp(a,b)   ((a).min==(b).min && (a).max==(b).max)
#define _StringEqOp(a,b)     (!strcmp(a,b))
#define _IntervalNeqOp(a,b)  ((a).min!=(b).min || (a).max!=(b).max)
```

```
#define _StringNeqOp(a,b)    strcmp(a,b)
#define _IntervalNeqOp(a,b)  ((a).min!=(b).min || (a).max!=(b).max)
#define _IntervalInOp(x,iv)  ((iv).min <= (x)  && (iv).max >= (x))
#define _MinOp(iv)           ((iv).min)
#define _MaxOp(iv)           ((iv).max)
#define _realmod(x,y)        f_fmod (x,y)
Int     _iExpOp (Int n, Int k);
Real    _riExpOp (Real x, Int k);
float f_pow (float, float);
#define _rExpOp(x,y)         f_pow(x,y)
#define p_IntervalEqOp(a,b)  _IntervalEqOp(a,b)
#define p_StringEqOp(a,b)    (!p_strcmp(a,b))
#define p_IntervalNeqOp(a,b) _IntervalNeqOp(a,b)
#define p_StringNeqOp(a,b)   p_strcmp(a,b)
#define p_IntervalInOp(x,iv) _IntervalInOp(x,iv)
#define p_MinOp(iv)          _MinOp(iv)
#define p_MaxOp(iv)          _MaxOp(iv)
#define p_realmod(x,y)       fp_fmod (x,y)
plural Int   p_iExpOp (plural Int n, plural Int k);
plural Real  p_riExpOp (plural Real x, plural Int k);
#define p_rExpOp(x,y)        fp_pow(x,y)
plural float fp_pow (plural float, plural float);  /* missing in mp_libm.h */
Interval    _IntIntervalOp   (Int  a, Int  b);
Interval2   _IntIntervalOp2  (Int2 a, Int2 b);
Interval1   _IntIntervalOp1  (Int1 a, Int1 b);
Realerval   _RealIntervalOp  (Real a, Real b);
plural Interval    p_IntIntervalOp   (plural Int  a, plural Int  b);
plural Interval2   p_IntIntervalOp2  (plural Int2 a, plural Int2 b);
plural Interval1   p_IntIntervalOp1  (plural Int1 a, plural Int1 b);
plural Realerval   p_RealIntervalOp  (plural Real a, plural Real b);

Interval _CanonicInterval2   (Interval2 i);
Interval _CanonicInterval1   (Interval1 i);
Interval _CanonicIntervalInt (Int i);
plural Interval p_CanonicInterval2   (plural Interval2 i);
plural Interval p_CanonicInterval1   (plural Interval1 i);
plural Interval p_CanonicIntervalInt (plural Int i);
#define _allslice    _IntIntervalOp (0,1<<30)
#define p__allslice  p_IntIntervalOp (0,1<<30)
}
```

This macro is invoked in definition 385.

The boolean operations on intervals are implemented as macros, as are the `MIN` and `MAX` operations. This has the advantage that there is no need for different variants, neither for the various interval types nor for sequential versus parallel execution. The interval construction operations must be implemented as functions. Also in this section are the equality operators on `STRING`, the exponentiation operators, and the `MOD` operator for `Real`.

RTS Operators[344] ≡ 344

```
{
Int _iExpOp (Int n, Int k)
{
  /* Exponentiation by integers via square and multiply: */
  Int result = n;
  if (k == 0)
    return (1);
```

```
  while (k > 1) {
    if ((k & 1) == 0) {
      result *= result; k >>= 1;   /* k even */
    }
    else {
      result *= n; k--;            /* k odd */
    }
  }
  return (result);
}

Real _riExpOp (Real x, Int k)
{
  /* Exponentiation by integers via square and multiply: */
  Real result = x;
  if (k == 0)
    return (1.0);
  while (k > 1) {
    if ((k & 1) == 0) {
      result *= result; k >>= 1;   /* k even */
    }
    else {
      result *= x; k--;            /* k odd */
    }
  }
  return (result);
}

plural Int p_iExpOp (plural Int n, plural Int k)
{
  /* Exponentiation by integers via square and multiply: */
  plural Int result = n;
  if (k == 0)
    return (1);
  while (k > 1) {
    if ((k & 1) == 0) {
      result *= result; k >>= 1;   /* k even */
    }
    else {
      result *= n; k--;            /* k odd */
    }
  }
  return (result);
}

plural Real p_riExpOp (plural Real x, plural Int k)
{
  /* Exponentiation by integers via square and multiply: */
  plural Real result = x;
  if (k == 0)
    return (1.0);
  while (k > 1) {
    if ((k & 1) == 0) {
      result *= result; k >>= 1;   /* k even */
    }
```

```
    else {
      result *= x; k--;            /* k odd */
    }
  }
  return (result);
}

#define _MakeIntervalOp(_it,_name,_t) \
  _it _name (_t a, _t b) { _it i; i.min=a; i.max=b; return(i); } \
  plural _it _cat2(p,_name) (plural _t a, plural _t b) \
    { plural _it i; i.min=a; i.max=b; return(i); }

_MakeIntervalOp (Interval,  _IntIntervalOp,  Int)
_MakeIntervalOp (Interval2, _IntIntervalOp2, Int2)
_MakeIntervalOp (Interval1, _IntIntervalOp1, Int1)
_MakeIntervalOp (Realerval, _RealIntervalOp, Real)

Interval _CanonicInterval1 (Interval1 i)
{
  return (_IntIntervalOp ((Int)i.min, (Int)i.max));
}

Interval _CanonicInterval2 (Interval2 i)
{
  return (_IntIntervalOp ((Int)i.min, (Int)i.max));
}

Interval _CanonicIntervalInt (Int i)
{
  return (_IntIntervalOp (i, i));
}

plural Interval p_CanonicInterval1 (plural Interval1 i)
{
  return (p_IntIntervalOp ((plural Int)i.min, (plural Int)i.max));
}

plural Interval p_CanonicInterval2 (plural Interval2 i)
{
  return (p_IntIntervalOp ((plural Int)i.min, (plural Int)i.max));
}

plural Interval p_CanonicIntervalInt (plural Int i)
{
  return (p_IntIntervalOp (i, i));
}
}
```

This macro is invoked in definition 387.

52.3 Topology change

This section contains topology change operations. Two of them implement language-visible functionality, the others are auxiliary functions for these or other topology changing operations.

rts.h Topology Change Operations[345] ≡ 345

```
{
 plural void a_MERGE__remote_connection (plural _network_D *net_D,
    plural _sint con_ls, plural _remote_connection *objs,
    _bool merge, _bool redistribute, _bool create_replicates);
 plural void DISCONNECT (plural _network_D* net_D,
    plural char* nd1, plural Interval slice1, plural _node_group_D* group_D1,
    int nd_D_offset1, int interf_D_offset1, int interf_offset1,
    plural char* nd2, plural Interval slice2, plural _node_group_D* group_D2,
    int nd_D_offset2, int interf_D_offset2, int interf_offset2,
    int consize1, int oe_offset1, int exists_offset1,
    int consize2, int exists_offset2);
 plural void REPLICATE_connection (plural _connection_D* con_D,
                                   plural Int into);
 plural void delete_connection_postprocessing (plural _connection_D *con_D,
    plural _Gptr *oe, int remote_descr_offset);
 plural void copy_connections (plural char *old_nd, plural char *nd,
    int ndsize, int node_D_offset, int interface_offset, int interface_D_offset,
    int consize, int con_D_offset, int oe_offset,
    plural _bint x0, plural _bint y0,
    plural _bint lxN, plural _bint lyN, plural _bint layer,
    plural _bool is_replicate0);
 plural void delete_connections (plural char *cons, int con_size,
    _sint con_ls, int oe_offset, int descr_offset, int oe_descr_offset);
 plural void reconnect1_connections (plural char *old_cons, plural char *cons,
    int consize, _sint old_con_ls, _sint con_ls,
    int oe_offset, int descr_offset, int oe_oe_offset);
 plural void reconnect_connections (plural char *cons, int con_size,
    _sint con_ls, int oe_offset, int descr_offset, int oe_oe_offset);
}
```

This macro is invoked in definition 385.

Most topology change operations are type-specific and must thus be implemented using templates. The exceptions are implemented in this module:

We need a simplified operation for the merge call to those interfaces that have remote connections attached. Nothing needs to be done to merge them, because the actual data merging occurs from the data end. But it is still necessary to implement the redistribution to be able to construct replicates using this procedure.

346 *RTS Topology Change Operations*[346] ≡

```
{
 plural void a_MERGE__remote_connection (plural _network_D *net_D,
    plural _sint con_ls, plural _remote_connection *objs,
    _bool merge, _bool redistribute, _bool create_replicates)
 {
   /* active set: all
   */
   _sint step, i;
   _sint repN = _sgl(net_D->repN);
   _bint lxN = _sgl(net_D->lxN),
         lyN = _sgl(net_D->lyN);
   _TRACE (4, ("MERGE_remoteCon (%x, %d/%d/%d)\n", (int)objs, (int)merge,
              (int)redistribute, (int)create_replicates));
   if (!create_replicates)
     return;  /* nothing to be done */
   _assert (!merge && redistribute && create_replicates);
   for (i = 0; i < con_ls; i++, objs++)
```

```
      ‘distribute this connection;

    ‘distribute this connection:
       plural _bool result_computed = net_D->exists && net_D->meI == 0 &&
                                      objs->_me_D.exists;
       if (ixproc < _S(lxN))
         ‘do distribution in y direction;
       ‘do distribution in x direction;
       objs->_oe.pe += (ixproc & ~_M(lxN)) + ((iyproc & ~_M(lyN)) << lxprocN);

    ‘do distribution in y direction:
       step = _S(lyN);
       while (step < yprocN) {
         if (iyproc + step < yprocN && result_computed && ‘y_neighbor_I < repN)
           ‘put y remote value;
         step <<= 1;
       }

    ‘put y remote value:
       ss_xsend (-step, 0, (plural void*)objs, (plural void*)objs,
                 sizeof(_remote_connection));
       xnetS[step].result_computed = true;

    ‘do distribution in x direction:
       step = _S(lxN);
       while (step < xprocN) {
         if (ixproc + step < xprocN && result_computed && ‘x_neighbor_I < repN)
           ‘put x remote value;
         step <<= 1;
       }

    ‘put x remote value:
       ss_xsend (0, step, (plural void*)objs, (plural void*)objs,
                 sizeof(_remote_connection));
       xnetE[step].result_computed = true;

    ‘x_neighbor_I:
       ((ixproc+step) >> lxN) + ((iyproc >> lyN) << (lxprocN-lxN))

    ‘y_neighbor_I:
       ((ixproc) >> lxN) + (((iyproc+step) >> lyN) << (lxprocN-lxN))

    }
    }
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

`DISCONNECT` implements the removal of connections under central control for all connection types. No template is necessary here, because the procedure does not need access to the data portion of the connection type — only to the descriptors. The procedure works in three phases: (1) Mark all connections in slice 1 as `shadow`, (2) find all connections in slice 2 whose opposite ends are marked as `shadow` and delete them, and (3) reset all connections in slice 1 that still exist from `shadow` to `existing`. This procedure uses the assumption that there is only one node virtualization layer per group which is always true in form 0.

RTS Topology Change Operations[347] ≡ 347

{

```
plural void DISCONNECT (plural _network_D* net_D,
   plural char* nd1, plural Interval slice1, plural _node_group_D* group_D1,
   int nd_D_offset1, int interf_D_offset1, int interf_offset1,
   plural char* nd2, plural Interval slice2, plural _node_group_D* group_D2,
   int nd_D_offset2, int interf_D_offset2, int interf_offset2,
   int consize1, int oe_offset1, int exists_offset1,
   int consize2, int exists_offset2)
{
  int i;
  plural _sint meI1, meI2;
  int con1_ls = _sgl(`interf1_D.con_ls),
      con2_ls = _sgl(`interf2_D.con_ls);
  plural char *cons;
  plural _Gptr oe;
  plural _realness ex;
  if (net_D->formA) {
    fprintf (stderr, "DISCONNECT not allowed for replicated networks\n");
    exit (11);
  }
  meI1 = `nd1_D.meI;
  meI2 = `nd2_D.meI;
  /* Phase 1: */
  if (`nd2_D.exists && _IntervalInOp (meI2, slice2)) {
    cons = _sgl(*(plural char* plural*)(nd2 + interf_offset2));
    for (i = 0; i < con2_ls; i++, cons += consize2)
      if (`con2_exists)
        `con2_exists = _shadow;
  }
  /* Phase 2: */
  if (`nd1_D.exists && _IntervalInOp (meI1, slice1)) {
    cons = _sgl(*(plural char* plural*)(nd1 + interf_offset1));
    for (i = 0; i < con1_ls; i++, cons += consize1)
      if (`con1_exists)
        `perhaps delete this connection;
  }
  /* Phase 3: */
  if (`nd2_D.exists && _IntervalInOp (meI2, slice2)) {
    cons = _sgl(*(plural char* plural*)(nd2 + interf_offset2));
    for (i = 0; i < con2_ls; i++, cons += consize2)
      if (`con2_exists)
        `con2_exists = _existing;
  }
  /* now invalidate the conN values in the interface descriptors: */
  `interf1_D.conN = invalid_conN;
  `interf2_D.conN = invalid_conN;

`perhaps delete this connection:
   oe = `con1_oe;
   ps_rfetch (oe.pe, oe.a+exists_offset2, (plural char*)&ex,
              sizeof (_realness));
   if (ex == _shadow) {
     ex = _nonexisting;
     sp_rsend (oe.pe, (plural char*)&ex, oe.a+exists_offset2,
               sizeof (_realness));
     `con1_exists = _nonexisting;
```

```
    }

  `nd1_D:
    *(plural _node_D*)(nd1+nd_D_offset1)

  `nd2_D:
    *(plural _node_D*)(nd2+nd_D_offset2)

  `interf1_D:
    *(plural _interface_D*)(nd1+interf_D_offset1)

  `interf2_D:
    *(plural _interface_D*)(nd2+interf_D_offset2)

  `con1_exists:
    *(plural _realness*)(cons+exists_offset1)

  `con2_exists:
    *(plural _realness*)(cons+exists_offset2)

  `con1_oe:
    *(plural _Gptr*)(cons+oe_offset1)

  }
  }
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

`REPLICATE_connection`, which only needs to manipulate the descriptor. Operations supporting some of the topology changing operations can be found in section 53.3.

RTS Topology Change Operations[348] ≡ 348

```
 {
  plural void REPLICATE_connection (plural _connection_D* con_D,
                                    plural Int into)
  {
    switch (into) {
      case 0:  con_D->exists = _nonexisting; return;  /* delete yourself */
      case 1:  return; /* null operation */
      default: /* replicate into many is illegal */
               fprintf (stderr, "REPLICATE Connection INTO %d impossible",
                        _sgl(into));
               exit (12);
    }
  }
  }
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

The rest of this section are internal (auxiliary) operations for topology change:

`delete_connection_postprocessing` must be called after a `REPLICATE connection INTO 0` to clean up the inconsistencies in the descriptor data structures: The `REPLICATE` merely sets the local `exists` marker to `nonexisting`; we now have to do the same for the remote `exists` marker and must invalidate the local and remote `conN` values in the interface descriptors, so that a call to `update_conI_conN` can later be triggered when necessary. Since the `exists` marker that was changed by the `REPLICATE connection INTO 0` is a faked one in the case of remote operations, we must set the actual marker to `nonexisting`, too. The procedure also checks that the network is in form 0 and exits the program with an error message if it is not.

349 *RTS Topology Change Operations*[349] ≡

```
{
 plural void delete_connection_postprocessing (plural _connection_D *con_D,
    plural _Gptr *oe, int remote_descr_offset)
 {
   plural _realness            ex = _nonexisting;
   plural _interface_D* plural remote_interf;
   plural _sint                new_conN = invalid_conN;
   if (con_D->boss->boss->boss->boss->formA) {
     fprintf (stderr, "REPLICATE con  called while network is replicated\n");
     exit (13);
   }
   /* set local and remote 'exists' value: */
   con_D->exists = _nonexisting; /* needed for remote operations! */
   sp_rsend (oe->pe, (plural char*)&ex,
             (plural char* plural)`remote exists address, sizeof (_realness));
   /* invalidate local conN: */
   con_D->boss->conN = invalid_conN; /* invalidate local conN */
   /* invalidate remote conN: */
   ps_rfetch (oe->pe, (plural char* plural)`remote bosspointer address,
              (plural char*)&remote_interf, sizeof (_interface_D*));
   sp_rsend (oe->pe, (plural char*)&new_conN,
             (plural char* plural)remote_interf + offsetof(_interface_D,conN),
             sizeof (new_conN));

 `remote exists address:
    oe->a + remote_descr_offset + offsetof (_connection_D,exists)

 `remote bosspointer address:
    oe->a + remote_descr_offset + offsetof (_connection_D,boss)
 }
}
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

`copy_connections` is a procedure used during network replication. For one old node virtualization layer it copies the connections of one interface to the corresponding new connection array, which must already be allocated, and sets the `oe` pointer of the old connection object to point to the new one (this is step 1 of the remote pointer fuddling described in section 30.4.2). The procedure uses several parameters that describe the overall node layout of the new node group.

350 *RTS Topology Change Operations*[350] ≡

```
{
 plural void copy_connections (plural char *old_nd, plural char *nd,
    int ndsize, int node_D_offset, int interface_offset, int interface_D_offset,
    int consize, int con_D_offset, int oe_offset,
    plural _bint x0, plural _bint y0,
    plural _bint lxN, plural _bint lyN, plural _bint layer,
    plural _bool is_replicate0)
 {
   /* This procedure assumes that the old connections have a correct
      global meI set and uses only connections from replicate 0.
      The numbering of connections within a node block is along x
      first, then y, and local index changing slowest.
```

```
      */
      int i;
      plural char* con = _sgl(*(plural char* plural*)(old_nd + interface_offset));
      _sint con_ls = _sgl(((plural _interface_D*)
                              (old_nd+interface_D_offset))->con_ls);
      plural _sint ndI;
      plural _bint x, y;
      plural _sint index;
      plural _Gptr target;
      _TRACE (2, ("copy_connections__ (%x,%x)\n", (int)old_nd, (int)nd));
      ndI = `old_nd _me_D.meI;
      if (is_replicate0 && `old_nd _me_D.exists)
        for (i = 0; i < con_ls; i++, con += consize)
          if (`con_D.exists)
            `copy this layer of connections;

    `copy this layer of connections:
        plural _sint conI = `con_D.meI;
        plural _bint my_lxN = router[ndI].lxN,
                     my_lyN = router[ndI].lyN;
        _lfold3(conI, my_lxN, my_lyN, x, y, index);
        target.pe = (router[ndI].x0 + x) + (router[ndI].y0 + y) * xprocN;
        target.a  = *(plural char* plural* plural)(nd + ndsize*router[ndI].layer +
                      interface_offset) + index*consize;
        `con_D.boss = &`nd interf_D;  /* destroys old boss descriptor ! */
        sp_rsend (target.pe, con, target.a, consize);
        /* now install 'Nachsendeantrag' (``where I have moved to''): */
        *(plural _Gptr*)(con+oe_offset) = target;

    `con_D:
       *(plural _connection_D*)(con+con_D_offset)

    `nd interf_D:
       *(plural _interface_D*)(nd + interface_D_offset)

    `old_nd _me_D:
       *(plural _node_D*)(old_nd + node_D_offset)

    }
    }
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

`delete_connections` marks all connections in a connection array as deleted. This includes setting the `exists` marker of the opposite end to `nonexisting`, too.

RTS Topology Change Operations[351] ≡ 351

```
   {
    plural void delete_connections (plural char *cons, int con_size,
       _sint con_ls, int oe_offset, int descr_offset, int oe_descr_offset)
    {
      /* for intra-group connections, the `exists' field is set to _nonexisting
         twice, but the procedure works correctly.
         active set: node blocks on which to delete
      */
      int i;
      plural _Gptr oe;
```

```
  plural _realness ex = _nonexisting;
  plural _interface_D* plural remote_interf;
  plural _sint new_conN = invalid_conN;
  for (i = 0; i < con_ls; i++, cons += con_size)
    if (‘con_D.exists) {
      /* set local and remote ’exists’ value: */
      ‘con_D.exists = ex;
      oe = *(plural _Gptr*)(cons+oe_offset);
      sp_rsend (oe.pe, (plural char*)&ex, oe.a+‘oe_exists_offset,
                sizeof(_realness));
      /* invalidate local and remote conN: */
      ‘con_D.boss->conN = invalid_conN; /* invalidate local conN */
      ps_rfetch (oe.pe, (plural char* plural)‘remote bosspointer address,
                 (plural char*)&remote_interf, sizeof (_interface_D*));
      sp_rsend (oe.pe, (plural char*)&new_conN,
                (plural char* plural)remote_interf +
                 offsetof(_interface_D,conN),
                sizeof (new_conN));
    }

‘con_D:
  *(plural _connection_D*)(cons+descr_offset)

‘oe_exists_offset:
   oe_descr_offset + offsetof(_connection_D,exists)

‘remote bosspointer address:
   oe.a + oe_descr_offset + offsetof(_connection_D,boss)
}
}
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

`reconnect_connections` implements the “follow `oe` pointer” part (step 2) of the remote connection pointer reorganization as described in section 30.4.2. `reconnect1_connections` does the equivalent for the node group extension and replicate node operations, where only a single group has to be reconnected so that intra-group connections are a special case.

352 *RTS Topology Change Operations*[352] ≡

```
{
 plural void reconnect_connections (plural char *cons, int con_size,
    _sint con_ls, int oe_offset, int descr_offset, int oe_oe_offset)
 {
   /* iterates over the NEW connections
      active set: blocks of existing nodes in replicate 0
   */
   int i;
   plural _Gptr oe, new_oe;
   for (i = 0; i < con_ls; i++, cons += con_size) {
     if (‘con_D.exists) {
       oe = *(plural _Gptr*)(cons+oe_offset);
 #ifdef NDEBUG
       ps_rfetch (oe.pe, oe.a+oe_oe_offset, cons+oe_offset, sizeof(_Gptr));
 #else
       ps_rfetch (oe.pe, oe.a+oe_oe_offset, (plural char*)&new_oe,
                  sizeof(_Gptr));
       *(plural _Gptr*)(cons+oe_offset) = new_oe;
```

```
#endif
    }
  }

`con_D:
  *(plural _connection_D*)(cons+descr_offset)
}


plural void reconnect1_connections (plural char *old_cons, plural char *cons,
   int consize, _sint old_con_ls, _sint con_ls,
   int oe_offset, int descr_offset, int oe_oe_offset)
{
  /* iterates over the NEW connections
     active set: replicate 0
  */
  int i;
  _sint cons_start = (_sint)cons,
        cons_end   = (_sint)(cons + con_ls*consize),
        old_cons_start = (_sint)old_cons,
        old_cons_end   = (_sint)(old_cons + old_con_ls*consize);
  plural _bool make_correction;
  plural _sint a;
  plural _Gptr oe,
               roe; /* remote oe: the oe of my oe */
  for (i = 0; i < con_ls; i++, cons += consize) {
    if (`con_D.exists) {
      oe = *(plural _Gptr*)(cons+oe_offset); /* copy of original oe pointer */
      ps_rfetch (oe.pe, oe.a+oe_oe_offset, (plural char*)&roe, sizeof(_Gptr));
      a = (plural _sint)roe.a;
      /* a is my remote's remote pointer.
         It normally points to my former me (i.e. into old_cons).
         For intra-layer connections, however, it already points into cons.
      */
      make_correction = true;
      if (!`a is in old_cons)
        `handle intra layer connection; /* may set make_correction = false */
      /* now send correct new oe data to my opposite end: */
      if (make_correction) {
        roe.pe = iproc;
        roe.a  = cons;
        sp_rsend (oe.pe, (plural char*)&roe, oe.a+oe_oe_offset, sizeof(_Gptr));
      }
    }
  }

`handle intra layer connection:
   /* There are two cases:
      1. this a is a `where I have moved to' pointer of an intra-group
         connection: roe shows where my actual oe has moved to,
         oe needs correction.
      2. This is an already corrected pointer. (Either from a
         previous virtualization layer at this interface or from
         a previously handled interface).
         Can be recognized by roe pointing into cons.
```

```
          Nothing at all needs to be done for this pointer.
      */
      if (`a is in cons) /* case 2 */
        make_correction = false;  /* skip correction: it has already been made */
      else                /* case 1 */
        *(plural _Gptr*)(cons+oe_offset) = oe = roe;

  `con_D:
    *(plural _connection_D*)(cons+descr_offset)

  `a is in old_cons:
    a >= old_cons_start && a < old_cons_end

  `a is in cons:
    a >= cons_start && a < cons_end

  }
  }
```

This macro is defined in definitions 346, 347, 348, 349, 350, 351, and 352.
This macro is invoked in definition 388.

52.4 Standard library

There are a number of functions that are defined in the run time system in order to be used as external functions or procedures in the user programs. Only declarations need to be given for these objects in a CuPit program. Most of them fall into one of two classes: output procedures for the builtin types and arithmetic functions not builtin into the language. Additional procedures allow access to command line arguments and timing measurement of program runs.

Here are the output functions for the basic types:

353 *rts.h Standard Library*[353] ≡

```
{
void pBool (Bool ME);
void pReal (Real ME);
void pString (String ME);
void pInt (Int ME);
void pInt1 (Int1 ME);
void pInt2 (Int2 ME);
void pRealerval (Realerval ME);
void pInterval (Interval ME);
void pInterval1 (Interval1 ME);
void pInterval2 (Interval2 ME);
plural void p_pBool (plural Bool ME);
plural void p_pReal (plural Real ME);
plural void p_pString (plural String ME);
plural void p_pInt (plural Int ME);
plural void p_pInt1 (plural Int1 ME);
plural void p_pInt2 (plural Int2 ME);
plural void p_pRealerval (plural Realerval ME);
plural void p_pInterval (plural Interval ME);
plural void p_pInterval1 (plural Interval1 ME);
plural void p_pInterval2 (plural Interval2 ME);
void pRealIO (plural Real **x, Interval which);
void pIntIO (plural Int **x, Interval which);
void pInt1IO (plural Int1 **x, Interval which);
```

```
void pInt2IO (plural Int2 **x, Interval which);
}
```

This macro is defined in definitions 353, 356, 362, and 365.
This macro is invoked in definition 385.

RTS Standard Library[354] ≡ 354

```
{
void pBool (Bool ME)
{  printf ("%c", ME ? 'T' : 'F');  }

void pReal (Real ME)
{  printf ("%g ", ME);  }

void pString (String ME)
{  printf ("%s", ME);  }

void pInt (Int ME)
{  printf ("%d ", ME);  }

void pInt1 (Int1 ME)
{  printf ("%d ", (int)ME);  }

void pInt2 (Int2 ME)
{  printf ("%d ", (int)ME);  }

void pRealerval (Realerval ME)
{  printf ("%g...%g ", ME.min, ME.max);  }

void pInterval (Interval ME)
{  printf ("%d...%d ", ME.min, ME.max);  }

void pInterval1 (Interval1 ME)
{  printf ("%d...%d ", (int)ME.min, (int)ME.max);  }

void pInterval2 (Interval2 ME)
{  printf ("%d...%d ", (int)ME.min, (int)ME.max);  }

plural void p_pBool (plural Bool ME)
{  p_printf ("%c", (plural int)(ME ? 'T' : 'F'));  }

plural void p_pReal (plural Real ME)
{  p_printf ("%g ", ME);  }

plural void p_pString (plural String ME)
{  p_printf ("%s", ME);  }

plural void p_pInt (plural Int ME)
{  p_printf ("%d ", ME);  }

plural void p_pInt1 (plural Int1 ME)
{  p_printf ("%d ", (plural int)ME);  }

plural void p_pInt2 (plural Int2 ME)
{  p_printf ("%d ", (plural int)ME);  }

plural void p_pRealerval (plural Realerval ME)
```

```
{  p_printf ("%g...%g ", ME.min, ME.max);  }

plural void p_pInterval (plural Interval ME)
{  p_printf ("%d...%d ", ME.min, ME.max);  }

plural void p_pInterval1 (plural Interval1 ME)
{  p_printf ("%d...%d ", (plural int)ME.min, (plural int)ME.max);  }

plural void p_pInterval2 (plural Interval2 ME)
{  p_printf ("%d...%d ", (plural int)ME.min, (plural int)ME.max);  }

void pRealIO (plural Real **x, Interval which)
{
  if (iproc >= which.min && iproc <= which.max)
    p_printf ("%.2f ", **x);
}

void pIntIO (plural Int **x, Interval which)
{
  if (iproc >= which.min && iproc <= which.max)
    p_printf ("%d ", **x);
}

void pInt1IO (plural Int1 **x, Interval which)
{
  if (iproc >= which.min && iproc <= which.max)
    p_printf ("%d ", (plural int)**x);
}

void pInt2IO (plural Int2 **x, Interval which)
{
  if (iproc >= which.min && iproc <= which.max)
    p_printf ("%d ", (plural int)**x);
}
}
```

This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.
This macro is invoked in definition 389.

To make the output operations for the basic types available to a CuPit user program, the following declarations have to be used:

355 *stdlib.nn Output Procedures*[355] ≡

```
{
PROCEDURE pBool (Bool CONST me) IS EXTERNAL;
PROCEDURE pReal (Real CONST me) IS EXTERNAL;
PROCEDURE pString (String CONST me) IS EXTERNAL;
PROCEDURE pInt (Int CONST me) IS EXTERNAL;
PROCEDURE pInt1 (Int1 CONST me) IS EXTERNAL;
PROCEDURE pInt2 (Int2 CONST me) IS EXTERNAL;
PROCEDURE pInterval (Interval CONST me) IS EXTERNAL;
PROCEDURE pInterval1 (Interval1 CONST me) IS EXTERNAL;
PROCEDURE pInterval2 (Interval2 CONST me) IS EXTERNAL;
PROCEDURE pRealIO (Real IO x; Interval CONST which) IS EXTERNAL;
PROCEDURE pIntIO (Int IO x; Interval CONST which) IS EXTERNAL;
PROCEDURE pInt1IO (Int1 IO x; Interval CONST which) IS EXTERNAL;
PROCEDURE pInt2IO (Int2 IO x; Interval CONST which) IS EXTERNAL;
}
```

This macro is invoked in definition 395.

The arithmetic functions in the library are a parameterized activation function (sigmoid as proposed by David Elliot, standard sigmoid, or gaussian) and its first and second derivatives; functions to compute the sign or absolute of a Real orInteger ; functions to compute the minimum or maximum of two Reals or Integers; and several transcendental functions (logarithms, square root, trigonometric functions).

rts.h Standard Library[356] ≡ 356

```
{
 Real activation (Real x, Int1 type);
 plural Real p_activation (plural Real x, plural Int1 type);
 Real activationPrime (Real x, Real actx, Int1 type);
 plural Real p_activationPrime (plural Real x, plural Real actx,
                                plural Int1 type);
 Real activationPrimePrime (Real x, Real actx, Real actpx, Int1 type);
 plural Real p_activationPrimePrime (plural Real x, plural Real actx,
                                     plural Real actpx, plural Int1 type);
 Real sqrtReal (Real x);
 Real logReal (Real x);
 Real log10Real (Real x);
 Real log2Real (Real x);
 Real sinReal (Real x);
 Real cosReal (Real x);
 Real tanReal (Real x);
 Real signReal (Real x);
 Real absReal (Real x);
 Real minReal (Real x, Real y);
 Real maxReal (Real x, Real y);
 Int signInt (Int x);
 Int absInt (Int x);
 Int minInt (Int x, Int y);
 Int maxInt (Int x, Int y);
 Int1 signInt1 (Int1 x);
 Int1 absInt1 (Int1 x);
 Int1 minInt1 (Int1 x, Int1 y);
 Int1 maxInt1 (Int1 x, Int1 y);
 Int2 signInt2 (Int2 x);
 Int2 absInt2 (Int2 x);
 Int2 minInt2 (Int2 x, Int2 y);
 Int2 maxInt2 (Int2 x, Int2 y);
 plural Real p_sqrtReal (plural Real x);
 plural Real p_logReal (plural Real x);
 plural Real P_log10Real (plural Real x);
 plural Real p_log2Real (plural Real x);
 plural Real p_sinReal (plural Real x);
 plural Real p_cosReal (plural Real x);
 plural Real p_tanReal (plural Real x);
 plural Real p_signReal (plural Real x);
 plural Real p_absReal (plural Real x);
 plural Real p_minReal (plural Real x, plural Real y);
 plural Real p_maxReal (plural Real x, plural Real y);
 plural Int p_signInt (plural Int x);
 plural Int p_absInt (plural Int x);
 plural Int p_minInt (plural Int x, plural Int y);
 plural Int p_maxInt (plural Int x, plural Int y);
 plural Int1 p_signInt1 (plural Int1 x);
```

```
plural Int1 p_absInt1 (plural Int1 x);
plural Int1 p_minInt1 (plural Int1 x, plural Int1 y);
plural Int1 p_maxInt1 (plural Int1 x, plural Int1 y);
plural Int2 p_signInt2 (plural Int2 x);
plural Int2 p_absInt2 (plural Int2 x);
plural Int2 p_minInt2 (plural Int2 x, plural Int2 y);
plural Int2 p_maxInt2 (plural Int2 x, plural Int2 y);
}
```

This macro is defined in definitions 353, 356, 362, and 365.
This macro is invoked in definition 385.

We implement three different activation functions: Type 0 is the soft symmetric sigmoid function $s_{soft}(x) = \frac{x}{1+|x|}$ as proposed by David Elliot with a range of -1 to 1. Type 1 is the standard sigmoid function $s_{std}(x) = \frac{1}{1+e^{-x}}$ with a range of 0 to 1. Type 2 is a gaussian activation function $g(x) = e^{-\frac{1}{2}x^2}$ with a range of 0 to 1. Type 0 is also the default type (used when a type other then 1 or 2 is requested). Additionally, we define type -1 to be the identity activation function.

357 *RTS Standard Library*[357] $\equiv$

```
{
Real activation (Real x, Int1 type)
{
  switch (type) {
    case 1:  return (`std sigmoid);
    case 2:  return (`gaussian);
    case -1: return (x);
    case 0:
    default: return (`soft sigmoid);
  }

`std sigmoid:
  x >  50.0 ?  1.0 :
  x < -50.0 ? -1.0 :
             1.0/(1.0+f_exp(-x))

`gaussian:
  x > 10.0 || x < -10.0 ? 1e-10 : f_exp (-0.5*x*x)

`soft sigmoid:
  x / (1.0 + (x < 0.0 ? -x : x))

}

plural Real p_activation (plural Real x, plural Int1 type)
{
  switch (type) {
    case 1:  return (`std sigmoid);
    case 2:  return (`gaussian);
    case -1: return (x);
    case 0:
    default: return (`soft sigmoid);
  }

`std sigmoid:
  x >  50.0 ?  1.0 :
  x < -50.0 ? -1.0 :
             1.0/(1.0+fp_exp(-x))
```

```
'gaussian:
  x > 10.0 || x < -10.0 ? 1e-10 : fp_exp (-0.5*x*x)

'soft sigmoid:
  x / (1.0 + (x < 0.0 ? -x : x))
}
}
```

This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.
This macro is invoked in definition 389.

The first derivative of the activation functions is the following: $s'_{soft}(x) = \frac{1}{(1+|x|)^2}$ and $s'_{std}(x) = \frac{e^{-x}}{(1+e^{-x})^2}$ and $g'(x) = -x\, e^{-\frac{1}{2}x^2}$.
These can be computed more efficiently, however, if the value of the activation itself is available. Since this is the case in learning programs, we make the activation available to the functions as an additional parameter. This simplifies the derivatives to $s'_{std}(x) = s_{std}(x)(1 - s_{std}(x))$ and $s'_{soft}(x) = \frac{s_{soft}(x)^2}{x^2}$ and $g'(x) = -x\, g(x)$.

RTS Standard Library[358] $\equiv$ 358

```
{
 Real activationPrime (Real x, Real actx, Int1 type)
 {
   Real help;
   switch (type) {
     case 1:  return (actx * (1-actx) + 1e-10); /* std sigmoid */
     case 2:  return (-x*actx);    /* gaussian */
     case -1: return (1);
     case 0:
     default: if (x < 1e-10 && x > -1e-10)
                return (1.0);
              else {
                help = actx/x;
                return (help*help);  /* soft sigmoid */
              }
   }
 }

 plural Real p_activationPrime (plural Real x, plural Real actx,
                                plural Int1 type)
 {
   plural Real help;
   switch (type) {
     case 1:  return (actx * (1-actx) + 1e-10); /* std sigmoid */
     case 2:  return (-x*actx);    /* gaussian */
     case -1: return (1);
     case 0:
     default: if (x < 1e-10 && x > -1e-10)
                return (1.0);  /* avoid division by zero */
              else {
                help = actx/x;
                return (help*help);  /* soft sigmoid */
              }
   }
 }
 }
```

This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.

This macro is invoked in definition 389.

The same game can be repeated for the second derivatives using the activation *and* its first derivative as auxiliary inputs. The second derivative of the activation functions is the following: $s''_{soft}(x) = \frac{-2|x|}{(1+|x|)^3\, x}$ and $s''_{std}(x) = s'_{std}(x)(1-2s_{std}(x))$ and $g''(x) = -e^{-\frac{1}{2}x^2} + x^2\, e^{-\frac{1}{2}x^2}$.
With the auxiliary inputs, the derivatives simplify to $s''_{soft}(x) = -2\, sign(x) \frac{s^{P}rime_{soft}(x)}{1+|x|}$ and $s''_{std}(x) = s'_{std}(x)(1-2s_{std}(x))$ and $g''(x) = (x^2-1)\, g(x)$.

359 *RTS Standard Library*[359] $\equiv$

```
{
 Real activationPrimePrime (Real x, Real actx, Real actpx, Int1 type)
 {
   switch (type) {
     case 1:  return (actpx * (1-2*actx));
     case 2:  return ((x*x-1)*actx);
     case -1: return (0);
     case 0:
     default: return (x < 0.0 ? 2*actpx/(1-x) : -2*actpx/(1+x));
   }
 }

 plural Real p_activationPrimePrime (plural Real x, plural Real actx,
                                     plural Real actpx, plural Int1 type)
 {
   switch (type) {
     case 1:  return (actpx * (1-2*actx));
     case 2:  return ((x*x-1)*actx);
     case -1: return (0);
     case 0:
     default: return (x < 0.0 ? 2*actpx/(1-x) : -2*actpx/(1+x));
   }
 }
}
```

This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.
This macro is invoked in definition 389.

Now for the very simple other functions of this module:

360 *RTS Standard Library*[360] $\equiv$

```
{
 Real sqrtReal (Real x)
 {  return (f_sqrt (x));  }

 Real logReal (Real x)
 {  return (f_log (x));  }

 Real log10Real (Real x)
 {  return (f_log10 (x));  }

 Real log2Real (Real x)
 {  return (f_log (x)*1.442695);  }

 Real sinReal (Real x)
 {  return (f_sin (x));  }

 Real cosReal (Real x)
```

```
{  return (f_cos (x));  }

Real tanReal (Real x)
{  return (f_tan (x));  }

Real signReal (Real x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

Real absReal (Real x)
{  return ((x < 0) ? -x : x);  }

Real minReal (Real x, Real y)
{  return (x < y ? x : y);  }

Real maxReal (Real x, Real y)
{  return (x > y ? x : y);  }

Int signInt (Int x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

Int absInt (Int x)
{  return ((x < 0) ? -x : x);  }

Int minInt (Int x, Int y)
{  return (x < y ? x : y);  }

Int maxInt (Int x, Int y)
{  return (x > y ? x : y);  }

Int1 signInt1 (Int1 x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

Int1 absInt1 (Int1 x)
{  return ((x < 0) ? -x : x);  }

Int1 minInt1 (Int1 x, Int1 y)
{  return (x < y ? x : y);  }

Int1 maxInt1 (Int1 x, Int1 y)
{  return (x > y ? x : y);  }

Int2 signInt2 (Int2 x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

Int2 absInt2 (Int2 x)
{  return ((x < 0) ? -x : x);  }

Int2 minInt2 (Int2 x, Int2 y)
{  return (x < y ? x : y);  }

Int2 maxInt2 (Int2 x, Int2 y)
{  return (x > y ? x : y);  }

plural Real p_sqrtReal (plural Real x)
{  return (fp_sqrt (x));  }
```

```
plural Real p_logReal (plural Real x)
{  return (fp_log (x));  }

plural Real p_log10Real (plural Real x)
{  return (fp_log10 (x));  }

plural Real p_log2Real (plural Real x)
{  return (fp_log (x)*1.442695);  }

plural Real p_sinReal (plural Real x)
{  return (fp_sin (x));  }

plural Real p_cosReal (plural Real x)
{  return (fp_cos (x));  }

plural Real p_tanReal (plural Real x)
{  return (fp_tan (x));  }

plural Real p_signReal (plural Real x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

plural Real p_absReal (plural Real x)
{  return ((x < 0) ? -x : x);  }

plural Real p_minReal (plural Real x, plural Real y)
{  return (x < y ? x : y);  }

plural Real p_maxReal (plural Real x, plural Real y)
{  return (x > y ? x : y);  }

plural Int p_signInt (plural Int x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

plural Int p_absInt (plural Int x)
{  return ((x < 0) ? -x : x);  }

plural Int p_minInt (plural Int x, plural Int y)
{  return (x < y ? x : y);  }

plural Int p_maxInt (plural Int x, plural Int y)
{  return (x > y ? x : y);  }

plural Int1 p_signInt1 (plural Int1 x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

plural Int1 p_absInt1 (plural Int1 x)
{  return ((x < 0) ? -x : x);  }

plural Int1 p_minInt1 (plural Int1 x, plural Int1 y)
{  return (x < y ? x : y);  }

plural Int1 p_maxInt1 (plural Int1 x, plural Int1 y)
{  return (x > y ? x : y);  }
```

```
plural Int2 p_signInt2 (plural Int2 x)
{  return ((x > 0) ? 1 : ((x == 0) ? 0 : -1));  }

plural Int2 p_absInt2 (plural Int2 x)
{  return ((x < 0) ? -x : x);  }

plural Int2 p_minInt2 (plural Int2 x, plural Int2 y)
{  return (x < y ? x : y);  }

plural Int2 p_maxInt2 (plural Int2 x, plural Int2 y)
{  return (x > y ? x : y);  }
}
```

This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.
This macro is invoked in definition 389.

To make the arithmetic operations defined above available to a CuPit user program, the following declarations have to be used:

stdlib.nn Arithmetic Functions[361] ≡ 361

```
{
 (* Activation types:
   -1: identity activation function
    0: Elliot's soft sigmoid -1...1   x/(1+abs(x))
    1: Standard sigmoid       0...1   1/(1+exp(-x))
    2: Gaussian               0...1   exp(-x*x/2)            *)
 Real FUNCTION activation (Real CONST x; Int1 CONST type) IS EXTERNAL;
 Real FUNCTION activationPrime (Real CONST x, actx; Int1 CONST type) IS EXTERNAL;
 Real FUNCTION activationPrimePrime (Real CONST x, actx, actpx;
                                     Int1 CONST type) IS EXTERNAL;
 Real FUNCTION sqrtReal (Real CONST x)      IS EXTERNAL;
 Real FUNCTION logReal (Real CONST x)       IS EXTERNAL;
 Real FUNCTION log10Real (Real CONST x)     IS EXTERNAL;
 Real FUNCTION log2Real (Real CONST x)      IS EXTERNAL;
 Real FUNCTION sinReal (Real CONST x)       IS EXTERNAL;
 Real FUNCTION cosReal (Real CONST x)       IS EXTERNAL;
 Real FUNCTION tanReal (Real CONST x)       IS EXTERNAL;
 Real FUNCTION signReal (Real CONST x)      IS EXTERNAL;  (* --> -1,0,1 *)
 Real FUNCTION absReal (Real CONST x)       IS EXTERNAL;  (* --> x or -x *)
 Real FUNCTION minReal (Real CONST x, y)    IS EXTERNAL;  (* --> x or y *)
 Real FUNCTION maxReal (Real CONST x, y)    IS EXTERNAL;  (* --> x or y *)
 Int  FUNCTION signInt (Int CONST x)        IS EXTERNAL;  (* see above... *)
 Int  FUNCTION absInt (Int CONST x)         IS EXTERNAL;
 Int  FUNCTION minInt (Int CONST x, y)      IS EXTERNAL;
 Int  FUNCTION maxInt (Int CONST x, y)      IS EXTERNAL;
 Int1 FUNCTION signInt1 (Int1 CONST x)      IS EXTERNAL;
 Int1 FUNCTION absInt1 (Int1 CONST x)       IS EXTERNAL;
 Int1 FUNCTION minInt1 (Int1 CONST x, y)    IS EXTERNAL;
 Int1 FUNCTION maxInt1 (Int1 CONST x, y)    IS EXTERNAL;
 Int2 FUNCTION signInt2 (Int2 CONST x)      IS EXTERNAL;
 Int2 FUNCTION absInt2 (Int2 CONST x)       IS EXTERNAL;
 Int2 FUNCTION minInt2 (Int2 CONST x, y)    IS EXTERNAL;
 Int2 FUNCTION maxInt2 (Int2 CONST x, y)    IS EXTERNAL;
 }
```

This macro is invoked in definition 395.

The `getArg` function accesses the numerical command line arguments stored into the `args` array by the `callCupit.c` front-end driver program. The arguments are numbered from 1 on. Non-existing arguments return the default value instead.

The `getName` function accesses the string command line arguments stored into the `names` array by the `callCupit.c` front-end driver program. The arguments are numbered from 1 on. Non-existing arguments return the default value instead.

362 *rts.h Standard Library*[362] ≡

```
{
Real getArg (Int argI, Real deflt);
String getName (Int argI, String deflt);
}
```

This macro is defined in definitions 353, 356, 362, and 365.
This macro is invoked in definition 385.

363 *RTS Standard Library*[363] ≡

```
{
extern visible int   _argsN;
extern visible float _args[];
extern visible int   _namesN;
extern visible int   _nameoffsets[];
extern visible char  _names[];

Real getArg (Int argI, Real deflt)
{
  return (argI > 0 && argI <= _argsN ? _args[argI-1] : deflt);
}

String getName (Int argI, String deflt)
{
  return (argI > 0 && argI <= _namesN ? _names+_nameoffsets[argI-1] : deflt);
}
}
```

This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.
This macro is invoked in definition 389.

364 *stdlib.nn Other Procedures*[364] ≡

```
{
Real FUNCTION getArg (Int CONST argI; Real CONST deflt) IS EXTERNAL;
String FUNCTION getName (Int CONST argI; String CONST deflt) IS EXTERNAL;
}
```

This macro is defined in definitions 364 and 367.
This macro is invoked in definition 395.

The `timerStart` function starts a timer that measures CPU time consumption of the CuPit program (including time waiting for page-faults) with a precision of about 20 ms. `timerValue` returns the time since the last call to `timerStart` as a Real value measured in seconds. `timerUnuseTicks` allows to tell the timer to subtract a certain amount of time, measured in ticks (80ns each), from the next `timerValue` it returns. These unuse times accumulate and are reset upon a `timerStart`.

365 *rts.h Standard Library*[365] ≡

```
{
void timerStart ();
Real timerValue ();
void timerUnuseTicks (unsigned ticks);
}
```

This macro is defined in definitions 353, 356, 362, and 365.
This macro is invoked in definition 385.

366 *RTS Standard Library*[366] ≡
```
{
static _work unuseTicks = 0;

void timerStart ()
{
  extern void mpCpuTimerStart (); /* #include <mp_time.h> gives errors */
  unuseTicks = 0;
  mpCpuTimerStart ();
}

Real timerValue ()
{
  /* mpCpuTimerElapsed() returns milliseconds: */
  extern unsigned int mpCpuTimerElapsed ();
  return ((Real)mpCpuTimerElapsed()*0.001 - (Real)unuseTicks*80e-9);
}

void timerUnuseTicks (unsigned ticks)
{
  unuseTicks += ticks;
}
}
```
This macro is defined in definitions 354, 357, 358, 359, 360, 363, and 366.
This macro is invoked in definition 389.

stdlib.nn Other Procedures[367] ≡ 367
```
{
PROCEDURE timerStart () IS EXTERNAL;
Real FUNCTION timerValue () IS EXTERNAL;
}
```
This macro is defined in definitions 364 and 367.
This macro is invoked in definition 395.

53 Internal operations

This section describes that part of the run time system that implements operations that are not directly visible in CuPit source programs. These are auxiliary functions for interprocessor communication, the dynamic memory allocation functions, and functions for computation (in particular address computation).

53.1 Communication operations

rts.h Communication Functions[368] ≡ 368
```
{
/* do xnet send operations with plural distance values: */
/* perform   op[(1<<ldist)-1].lval = val   for plural ldist: */
#define xsend(op, ldist, lval, val) \
  { int __i; \
    for (__i = lxprocN; __i >= 0; __i--) \
      if ((ldist) == __i) \
        op[(1<<__i)-1].lval = (val); \
  }
```

```
/* perform   ss_xfetch or ss_rfetch  whichever is cheaper: */
#define ss_fetchx(dx, src, dest, size) \
  { if (dx > 32) \
      ss_rfetch (iproc + (dx), src, dest, size); \
    else \
      ss_xfetch (0, (dx), src, dest, size); \
  }

#define ss_fetchy(dy, src, dest, size) \
  { if (dy > 1) \
      ss_rfetch (iproc + ((dy)<<lxprocN), src, dest, size); \
    else \
      ss_xfetch (-(dy), 0, src, dest, size); \
  }

plural void ss_xsendc (plural _bint ldist_y, plural _bint ldist_x,
                       plural void* src_and_dest, _sint nbytes,
                       plural _bool got_it);
}
```

This macro is invoked in definition 385.

xsend implements xnet send, copy, or pipe operations with plural distance values which must be powers of two and are given in logarithmic form.

ss_fetchx and **ss_fetchy** perform a horizontal or vertical fetch operation with singular operand addresses and singular distance by either an xnet or a router operation, whichever is cheaper.

ss_xsendc broadcasts a number of bytes to all PEs in the southeast rectangle of each PE that has the parameter **got_it** true. These rectangles may not overlap.

369 *RTS Communication Operations*[369] ≡

```
{
plural void ss_xsendc (plural _bint ldist_y, plural _bint ldist_x,
                       plural void* src_and_dest, _sint nbytes,
                       plural _bool got_it)
{
  /* Broadcast nbytes bytes from the set of active PEs with got_it == true
     to the rectangle of size '_S(ldist_x)' times '_S(ldist_y)' PEs that has
     a got_it == true PE as its upper left (Northwest) corner.
     Overlapping of the rectangles is not allowed.
     active set: at least all source and target PEs
  */
  int ldist;  /* NOT plural! */
  int transmitted;
  plural char *a;
 plural int test;
  'distribute ldist values;
  test = (plural int)*(plural char*)src_and_dest;

'distribute ldist values:
   int i;
   if (!got_it)
     ldist_x = ldist_y = 255; /* set to unique and impossible value */
   if (got_it) {
     for (i = lxprocN; i > 0; i--)
       if (ldist_x == i) {
         xnetcE[_M(i)].ldist_x = ldist_x;
```

```
          xnetcE[_M(i)].ldist_y = ldist_y;
          ‘send data east;
        }
    }
    if (ldist_x != 255) {
      for (i = lyprocN; i > 0; i--)
        if (ldist_y == i) {
          xnetcS[_M(i)].ldist_x = ldist_x;
          xnetcS[_M(i)].ldist_y = ldist_y;
          ‘send data south;
        }
    }

 ‘send data east:
    for (transmitted = 0, a = src_and_dest;
         transmitted + sizeof(long long) <= nbytes;
         transmitted += sizeof(long long), a += 8)
      xnetcE[_M(i)].*(plural long long*)a = *(plural long long*)a;
    if (transmitted + sizeof (int) <= nbytes) {
      xnetcE[_M(i)].*(plural int*)a = *(plural int*)a;
      transmitted += sizeof (int);
    }
    if (transmitted + sizeof (_sint) <= nbytes) {
      xnetcE[_M(i)].*(plural _sint*)a = *(plural _sint*)a;
      transmitted += sizeof (_sint);
    }
    if (transmitted + sizeof (_bint) <= nbytes) {
      xnetcE[_M(i)].*(plural _bint*)a = *(plural _bint*)a;
      transmitted += sizeof (_bint);
    }
    _assert (transmitted == nbytes);

 ‘send data south:
    for (transmitted = 0, a = src_and_dest;
         transmitted + sizeof(long long) <= nbytes;
         transmitted += sizeof(long long), a += 8)
      xnetcS[_M(i)].*(plural long long*)a = *(plural long long*)a;
    if (transmitted + sizeof (int) <= nbytes) {
      xnetcS[_M(i)].*(plural int*)a = *(plural int*)a;
      transmitted += sizeof (int);
    }
    if (transmitted + sizeof (_sint) <= nbytes) {
      xnetcS[_M(i)].*(plural _sint*)a = *(plural _sint*)a;
      transmitted += sizeof (_sint);
    }
    if (transmitted + sizeof (_bint) <= nbytes) {
      xnetcS[_M(i)].*(plural _bint*)a = *(plural _bint*)a;
      transmitted += sizeof (_bint);
    }
    _assert (transmitted == nbytes);

 }
 }
```

This macro is invoked in definition 390.

53.2 Memory allocation

The memory allocation functions used in the code generated by the CuPit compiler are just wrappers around the `p_malloc` and `p_free` functions of the MasPar library.

370 *rts.h Memory Allocation*[370] ≡

```
{
 plural void  _initgetmem (int size_of_dynamic_memory);
 plural void* _getmem (int size, _bool zero_out);
 plural void  _freemem (plural void *a);
}
```

This macro is invoked in definition 385.

We implement the functions `initgetmem` to initialize the dynamic memory allocation, `getmem` to allocate a block of storage on each active PE (which has the same size and address on each PE), and `freemem` to release allocated memory. If not enough memory is available, `getmem` generates an error message and terminates the program. An option parameter allows to tell `getmem` to initialize the allocated memory with all zeroes.

If the symbol `memorytrace` is defined, this module also contains the functions `writetrace` and `tracemem` which write a complete report of all memory allocations into the file given in `TRACEFILE`. For each point in time, this report contains all segments of memory currently allocated with their locical and physical size and physical address. This report can be plotted with `gnuplot` to visualize usage of dynamically allocated memory. Writing of the report is suppressed if the `tracelevel` of the CuPit progam is zero.

371 *RTS Memory Allocation*[371] ≡

```
{
 #define memorytrace
 #ifdef memorytrace

 extern int _tracelevel;  /* from CuPit program */

 #define TRACEFILE "memtrace"
 #define SEGS 150

 static FILE *mt;  /* memory trace file */
 static _sint sStart[SEGS],  /* start of reserved memory segments */
              sSize[SEGS];   /* size of allocated memor segs (0 -> unused) */
 static int   segsN = 0,     /* number of currently allocated memory segments */
              highestI = -1, /* index of highest used segment entry */
              memtime = 0;   /* number of current timestep */

 static void writetrace ()
 {
    /* brute force approach:
       for each segment currently reserved, we write one line in
       gnuplot errorbar style: x y ylow yhigh
         x is just a running number (getmem time axis)
         y is addr+size-1 of the segment (logical end address)
         ylow = addr
         yhigh = end of segment in real memory consumption:
                 addr + (1<<log2(size+4))
    */
    int j;
    for (j = 0; j <= highestI; j++)
      if (sSize[j] != 0) {
        fprintf (mt, "%d %d %d %d\n", memtime, sStart[j]+sSize[j]-1,
```

```
                sStart[j], sStart[j] + (1<<_log2(sSize[j]+4)));
     }
   fflush (mt);
}

static void tracemem (plural void* addr, int size, _bool free)
{
  /* Announces that a memory segment at 'addr' was free'd (if free = true) or
     that a memory segment of size 'size' has been reserved at 'addr'.
  */
  int i;
  memtime++;
  if (_tracelevel == 0)
    return;
  if (free)
    `release segment;
  else
    `reserve segment;

`release segment:
   for (i = 0; i <= highestI; i++)
     if (sStart[i] == (int)addr) {
       sSize[i] = 0;  /* mark as released */
       segsN--;
       if (i == highestI)
         highestI--;
       writetrace ();
       return;
     }
   _assert (false);  /* segment was not found! */

`reserve segment:
   if (size == 0)
     return;
   for (i = 0; i < SEGS; i++)
     if (sSize[i] == 0) {
       sStart[i] = (int)addr;
       sSize[i] = (_sint)size;
       segsN++;
       if (i > highestI)
         highestI = i;
       writetrace ();
       return;
     }
   _assert (false);  /* segment could not be marked: too many segments */
}
#endif

plural void _initgetmem (int size_of_dynamic_memory)
{
  /* nothing to be done */
#ifdef memorytrace
  if (_tracelevel == 0)
    return;
  mt = fopen (TRACEFILE, "w");
```

```
  if (mt == 0) {
    fprintf (stderr, "couldn't write `%s'\n", TRACEFILE);
    exit (21);
  }
#endif
}

plural void* _getmem (int size, _bool zero_out)
{
  plural void* r;
  if (size == 0)
    return (0);
  r = p_malloc (size);
  if (r == 0) {
    fprintf (stderr, "\n*** not enough PE memory (requested: %d bytes) ***\n",
             size);
    exit (31);
  }
  if (zero_out)
     p_memset (r, (plural int)0, (plural size_t)size);
#ifdef memorytrace
  tracemem (r, size, false);
#endif
  return (r);
}

plural void _freemem (plural void *a)
{
  _assert ((int)a < 16384);  /* pointers beyond memory size are garbage! */
  if (a != 0) {
    p_free (a);
#ifdef memorytrace
    tracemem (a, 0, true);
#endif
  }
}
}
```

This macro is invoked in definition 391.

53.3 Computation

For the computation of size and placement of segments, blocks, and connections we need a number of routines that are defined in this section.

372 *rts.h Address Computations*[372] ≡

```
{
#define _S(la) (1<<(la))       /* Shift: maps log2(a) --> a */
#define _M(la) ((1<<(la))-1)   /* Mask:  maps log2(a) --> a-1 */

#define _lfold2(n,lxsize,lysize,x,y) {\
  x = (n) & _M(lxsize); \
  y = (n) >> (lxsize); \
  _assert ((y) < _S(lysize)); }

#define _lfold3(n,lxsize,lysize,x,y,i) {\
```

```
  x = (n) & _M(lxsize); \
  y = (n) >> (lxsize); \
  i = (y) >> (lysize); \
  y &= _M(lysize); }

#define _fold2(n,xsize,ysize,x,y) {\
  x = (n) % (xsize); \
  y = (n) / (xsize); \
  _assert ((y) < (ysize)); }

#define _fold3(n,lxsize,lysize,x,y,i) {\
  x = (n) % (xsize); \
  y = (n) / (xsize); \
  i = (y) / (ysize); \
  y %= (ysize); }

#define _unlfold2(lxsize,x,y)           ((x) + ((y) << (lxsize)))
#define _unlfold3(lxsize,lysize,x,y,i)  ((x) + ((y) << (lxsize)) + \
                                         ((i) << ((lxsize) + (lysize))))
#define _unfold2(xsize,x,y)           ((x) + (y) * (xsize))
#define _unfold3(xsize,ysize,x,y,i)  ((x) + (y) * (xsize) + \
                                        (i) * (xsize) * (ysize))

_bint        _log2 (int x);
plural _bint _p_log2 (plural int x);

plural _sint compute_block_sizesA (plural _work work, _sint nodesN,
    _sint newnodesN, int segmentsize);
plural void compute_block_size0 (plural _network_D* net_D,
    plural _node_group_D *group_D, _bint *lxblocksize, _bint *lyblocksize);
plural void compute_block_layout (plural _sint size, _sint nodesN,
    _sint newnodesN, _bint segment_lxN, _bint segment_lyN, _sint max_lblocksize,
    plural _bint *x0, plural _bint *y0, plural _bint *lxN, plural _bint *lyN,
    plural _bint *layer, _bint *layersN);
plural void update_conI_conN (plural char *cons, int con_size,
    int descr_offset, plural _interface_D *interf_D);
_work wpc_sum (plural _interface_D *interf_D);
}
```

This macro is invoked in definition 385.

`fold2` and `lfold2` convert a number `n` into two indices `x,y` into a two dimensional array of size `(xsize,ysize)`, where `x` changes fastest; for `lfold2`, the sizes of the array dimensions must be powers of two and are given in logarithmic form. `fold3` and `lfold3` do the same for a three-dimensional array with third index `i` changing slowest. `unfold2`, `unlfold2`, `unfold3`, and `unlfold3` are the inverse operations and return the value of `n` they compute. The value of `lysize` is not actually used in `fold2` and `lfold2` but is present for a sanity check.

The integral logarithm function `log2` returns the *ceiling* of the base 2 logarithm, i.e., the integer that is the exponent of the smallest power of two that is larger than or equal to the argument.

`compute_block_sizesA` computes the size of each node block from the work performed by that node. The node sizes implicitly determine the number of node virtualization layers used. `compute_block_size0` computes the common block size of the nodes of a form 0 node group.

`compute_block_layout` computes a layout of node blocks over several node virtualization layers from the node block sizes. The layout is described by giving for each node block its upper left corner, x and y size, and virtualization layer.

`compute_local_rtiwork_conI_conN` counts the number of actually existing connections of one node shadow interface in each node shadow of one node virtualization layer and locally sums the total work actually performed by them. During counting, the `meI` field in each connection interface is set properly. The function is used only in the rti version of the generated code where real measurements of work are available in the connection descriptor of each single connection. The plain and optimized versions use `compute_local_work_conI_conN` instead, which also counts the connections (and sets `meI`) but computes the work by multiplication with a `work_per_connection` factor that is given as a parameter (taken from the interface descriptor where it is stored for each interface of each node group). Both, `compute_local_rtiwork_conI_conN` and `compute_local_work_conI_conN` compute the correct value for the `meI` field in each connection descriptor in a *local* sense, i.e., "I am the n^{th} connection in the local connection array"; the values computed for work and number of connections are also local.

`update_conI_conN` computes the global value of the `meI` field in each connection descriptor, i.e, "I am the n^{th} connection at this interface in the node block". It assumes that each `meI` is set correctly in the local sense mentioned above and that the number of local connections is given as a parameter. In addition, this procedure counts the total number of connections at each node interface (across all shadows).

The procedure `wpc_sum` is called for one interface in one node virtualization layer; it computes the sum of the `wpc` values of the interface at the existing nodes.

373 *RTS Address Computations*[373] $\equiv$

```
{
 _bint _log2 (int x)
 {
   _bint result = 0;
   int   i = x;
   while (i > 0) {
     i >>= 1;
     result++;
   }
   return ((1<<(result-1)) == x ? result-1 : result);
 }

 plural _bint _p_log2 (plural int x)
 {
   plural _bint result = 0;
   plural int   i = x;
   while (i > 0) {
     i >>= 1;
     result++;
   }
   return ((1<<(result-1)) == x ? result-1 : result);
 }
}
```

This macro is defined in definitions 373 and 374.
This macro is invoked in definition 392.

For `compute_block_sizesA` we use the following strategy: The blocks are sized so as to fill the segment at most exactly once (if `newnodesN` is 0) or twice (otherwise). "Suggested" node block size is proportional to work. This value is then rounded up to the next higher power of two. These values usually overfill the segment. This can be corrected by rounding some of those block sizes (that were rounded up the most) down instead of up. To do this, we determine by binary search a cutoff threshold on the ratio of power of two block size to suggested block size (which is between 1 and 2), that is the highest that cuts of enough much-rounded-up blocks. Some of the blocks that are above this threshold are then rounded down (to a power of two) in size. We round down only as many blocks as necessary, from lowest block numbers to highest.

374 *RTS Address Computations*[374] $\equiv$

```
{
 plural _sint compute_block_sizesA (plural _work work, _sint nodesN,
                                    _sint newnodesN, int segmentsize)
 {
   _sint oldnodesN = nodesN - newnodesN;
   plural _work work1 = iproc < oldnodesN ? work : 0,
                work2 = iproc >= oldnodesN ? work : 0;
   _work totalwork1 = reduceAdd64u (work1),
         totalwork2 = ‘layer 2 totalwork;
   plural _sint result1 = 0, result2 = 0;
   plural float PEshould, ratio;
   plural _sint PEup, PEdown, PEtry, diffs;
   _sint        sum, oldsum, mustsave;
   _TRACE (1, ("compute_block_sizesA (%d, %d/%d, %d/%d, %d)\n",
               (int)proc[0].work, (int)totalwork1, (int)nodesN,
               (int)totalwork2, (int)newnodesN, segmentsize));
   if (nodesN == 0)
     return;
   if (oldnodesN > segmentsize || newnodesN > segmentsize) {
     fprintf (stderr, "Too many nodes for segment. Use fewer replicates\n");
     exit (32);
   }
   if (iproc < oldnodesN)
     ‘compute first layer layout;
   if (iproc >= oldnodesN && iproc < nodesN)
     ‘compute second layer layout;
   _TRACE (2, ("blocksizes: newnd0: %d, newnd1: %d, newnd2: %d\n",
               (int)proc[oldnodesN].result2, (int)proc[oldnodesN+1].result2,
               (int)proc[oldnodesN+2].result2));
   return (result1+result2);

 ‘compute first layer layout:
    plural char seg = 0;
    if (totalwork1 == 0)
      PEshould = (float)segmentsize / (float)oldnodesN;
    else
      PEshould = (plural float)(work1*segmentsize) / (float)totalwork1;
    ‘compute cutoff ratio threshold;
    /* now use the ratio only on as many blocks as necessary, beginning at 0 */
    diffs = PEup - PEtry;  /* differences when rounding down */
    proc[oldnodesN-1].seg = 1;
    result1 = scanAdd16u (diffs, seg);
    if (result1-diffs >= mustsave)
      PEtry = PEup;
    result1 = PEtry;

 ‘compute cutoff ratio threshold:
    float upper, lower, limit;
    int   stepsN;
    _bool finished;
    if (PEshould < 1.0)
       PEshould = 1.0;
    PEup = (plural _sint)_S(_p_log2((plural int)(PEshould+0.99)));
    PEdown = PEup == 1 ? 1 : PEup>>1;
    if (_tracelevel >= 1) {
```

```
      printf ("PEshould/up/down:");
      p_printf ("  %g/%d/%d", PEshould, PEup, PEdown);
      printf ("  (sums:)%g/%d/%d\n", reduceAddf(PEshould),
              reduceAdd16u(PEup), reduceAdd16u(PEdown));
    }
    sum = reduceAdd16u (PEdown);
    _assert (sum <= segmentsize); /* no problem if this happens, because: */
    if (sum > segmentsize) {  /* beware of the ghosts of float arithmetic! */
      PEup = (plural _sint)_S(_p_log2((plural int)PEshould));
      PEdown = PEup == 1 ? 1 : PEup>>1;
    }
    /* we may still have a problem due to too many PEup==1: */
    while (reduceAdd16u (PEdown) > segmentsize)
      PEdown = PEdown == 1 ? 1 : PEdown>>1;
    mustsave = reduceAdd16u (PEup) - segmentsize;
    ratio = (plural float)PEup / PEshould; /* is in [0.5...2) */
    upper = 2.0;
    lower = 0.5;
    limit = 1.5;
    sum = 0;
    stepsN = 0;
    finished = false;
    PEtry = PEup;
    /* go for the highest ratio limit that does not overfill the segment */
    while (!finished && mustsave != 0) {
      stepsN++;
      PEtry = ratio >= limit ? PEdown : PEup;
      oldsum = sum;
      sum = reduceAdd16u (PEtry);
      finished = (sum <= segmentsize &&
                  ((oldsum > segmentsize && stepsN >= 4) ||
                  sum == segmentsize)) ||
                 stepsN >= 10;
      if (sum > segmentsize)  /* limit too high, reduce it */
        upper = limit;
      else  /* limit too low, increase it again */
        lower = limit;
      limit = (upper+lower)/2.0;
    }
    if (sum > segmentsize)
      PEtry = ratio >= lower ? PEdown : PEup;

  `compute second layer layout:
    plural char seg = 0;
    if (totalwork2 == 0)
      PEshould = (float)segmentsize / (float)newnodesN;
    else
      PEshould = (plural float)(work2*segmentsize) / (float)totalwork2;
    `compute cutoff ratio threshold;
    /* now use the ratio only on as many blocks as necessary, beginning at 0 */
    diffs = PEup - PEtry;  /* differences when rounding down */
    proc[nodesN-1].seg = 1;
    result2 = scanAdd16u (diffs, seg);
    if (result2-diffs >= mustsave)
      PEtry = PEup;
```

```
    result2 = PEtry;

  ‘layer 2 totalwork:
    newnodesN ? reduceAdd64u (work2) : 0
  }


  plural void compute_block_size0 (plural _network_D* net_D,
      plural _node_group_D *group_D, _bint *lxblocksize, _bint *lyblocksize)
  {
    /* precondition:  group_D->nodesN set, net_D set
                      group_D->boss == net_D, !net_D->formA
          (the network segment size is expected to be procN, but we don’t
           rely on this)
       postcondition: group_D->localsize set,
                      *lxblocksize and *lyblocksize set
       strategy: The blocksize is chosen so as to fill the segment just once.
    */
    int    segmentsize, blocksize;
    _bint  lsegmentsize, lblocksize;
    _sint  nodesN = _sgl(group_D->nodesN);
    _assert (!net_D->formA);
    _TRACE (2, ("compute_block_size0 (%x, %x)\n", (int)net_D, (int)group_D));
    lsegmentsize = _sgl (net_D->lxN + net_D->lyN);
    segmentsize = _S(lsegmentsize);
    blocksize = segmentsize / nodesN;
    if (blocksize == 0) {
      blocksize  = 1;
      lblocksize = 0;
    }
    else {
      lblocksize = _log2 (blocksize - (blocksize>>1) + 1);  /* rounds down */
      if ((nodesN << lblocksize) > segmentsize)  /* the +1 above may have */
        lblocksize--;                            /* provoked rounding up! */
      blocksize = _S(lblocksize);
    }
    *lyblocksize = lblocksize >> 1;
    *lxblocksize = lblocksize - *lyblocksize;
    group_D->localsizeN = nodesN / ‘nr of blocks +
                 (nodesN % ‘nr of blocks != 0);
    _TRACE (2, ("lxblocksize=%d, lyblocksize=%d, group_D->localsize=%d\n",
                (int)*lxblocksize, (int)*lyblocksize,
                (int)_sgl(group_D->localsizeN)));

  ‘nr of blocks:
    _S(lsegmentsize - lblocksize)

  }


  plural void compute_block_layout (
      plural _sint  size,    /* on proc[i] = size of node block of node i */
      _sint         nodesN,  /* number of nodes to consider total */
      _sint         newnodesN,      /* number of nodes for layer 2 */
      _bint         segment_lxN,    /* log2 of (maximal ixproc to use + 1) */
```

```
    _bint         segment_lyN,    /* dito iyproc */
    _sint         max_lblocksize, /* log2 of maximum possible blocksize */
    plural _bint *x0,
    plural _bint *y0,      /* upper left corner of node block of node i */
    plural _bint *lxN,     /* on proc[i]: */
    plural _bint *lyN,     /* log size of node block of node i */
    plural _bint *layer,   /* on proc[i] = virtualization layer of node i */
    _bint        *layersN) /* highest virtualization layer used */
{
  /* this function computes a linear block layout, i.e., virtualization
     layer i contains all nodes k[i] to k[i+1]-1 with k[i] < k[i+1]
     for all i, k[0] = 0, k[layersN] = nodesN-1, and layersN minimal.
     The layout computed is for segment 0 only which is assumed to be at
     the upper left of the PE array (0...segment_xN-1, 0...segment_yN-1)
     We use only a restricted version of the algorithm, which knows
     which nodes to put into which layer in advance (layer 1: 0 to
     nodesN-newnodesN-1, layer 2: all others). The general version is
     available by #define general_cbl

  */
  plural int cum_size;  /* on proc[i] = cumulated block size of nodes 0..i */
  int low, high;        /* current lowest/highest node number to consider */
  plural _bool is_free; /* layout of current virtualization layer: each PE is
                           marked as free(=true) or occupied(=false) */
  int   l;              /* number of current virtualization layer */
  int   lbs,            /* current log block size considered */
        lxbs, lybs;     /* current log x and y block size considered */
  int   segment_size = _S(segment_lxN + segment_lyN);
  plural _bool within_segment = ixproc < _S(segment_lxN) &&
                                iyproc < _S(segment_lyN);
  _assert (nodesN < procN);
  _TRACE (2, ("compute_block_layout (%d, %d, %d)\n", (int)proc[0].size,
            (int)nodesN, (int)max_lblocksize));
  low  = 0;
  l = 0;  /* index of current virtualization layer */
  #ifdef general_cbl
    if (iproc < nodesN)
      cum_size = scanAdd32 ((plural int)size, 0); /* unsegmented scan */
  #endif
  while (low < nodesN) {
    `find high;
    `layout one virtualization layer;
    #ifdef general_cbl
      cum_size -= proc[high].cum_size;
    #endif
    low = high + 1;
    l++;
  }
  _TRACE (2, ("nd2 at %d/%d  logsize %d/%d  layer %d\n", (int)proc[2].*x0,
            (int)proc[2].*y0, (int)proc[2].*lxN, (int)proc[2].*lyN,
            (int)proc[2].*layer));
  *layersN = l;

`find high:
   #ifdef general_cbl
```

```
      plural _bool fits = false;
      if (iproc < nodesN && cum_size <= segment_size)
        fits = true;
      if (!fits)
        high = selectFirst() - 1;  /* the last one that fits is 'high' */
    #else
      high = (low == 0) ? nodesN - newnodesN - 1 : nodesN -1;
    #endif

  `layout one virtualization layer:
    is_free = true;  /* all PEs are initially free on a virtualization layer */
    for (lbs = max_lblocksize; lbs >= 0; lbs--)
      `layout blocks of one size;

  `layout blocks of one size:
    plural short free_blocks_enum = procN+1;/* enumeration of free blocks */
    plural short relevant_nodes_enum = -1;  /* enumeration of nodes to fit */
    plural _sint relevant_nodesN;           /* number of nodes to fit */
    lybs = lbs >> 1;
    lxbs = lbs - lybs;
    `find mark and store new blocks to use;

  `find mark and store new blocks to use:
    plural _bint block_x0, block_y0;  /* 'meet data' of blocks */
    /* find and count relevant nodes: */
    if (iproc >= low && iproc <= high && size == _S(lbs)) {
      relevant_nodes_enum = enumerate();
    }
    relevant_nodesN = reduceMax16 (relevant_nodes_enum) + 1;
    /* find blocks and put meeting data: */
    if (within_segment && is_free &&
        (ixproc & _M(lxbs)) == 0 && (iyproc & _M(lybs)) == 0) {
      free_blocks_enum = enumerate();
      if (free_blocks_enum < relevant_nodesN) {
        is_free = false;
        router[free_blocks_enum].block_x0 = ixproc;
        router[free_blocks_enum].block_y0 = iyproc;
      }
    }
    /* mark blocks as non-free: */
    ss_xsendc (lybs, lxbs, &is_free, sizeof (is_free),
               free_blocks_enum != -1 && free_blocks_enum < relevant_nodesN);
    /* fetch data to relevant nodes: */
    if (relevant_nodes_enum != -1) {
      *x0 = router[relevant_nodes_enum].block_x0;
      *y0 = router[relevant_nodes_enum].block_y0;
      *lxN = lxbs;
      *lyN = lybs;
      *layer = 1;
    }

  }


  plural void update_conI_conN (plural char *cons, int con_size,
```

```
                              int descr_offset, plural _interface_D *interf_D)
{
  /* This procedure is called for one interface in one node
     virtualization layer in replicate 0.
     It computes the correct values of conI
     in each connection and of conN in the interface descriptor.
     May do nothing if nothing is necessary.
     1. compute local conI values in the connection objects.
     2. compute a prefix sum over local conN in each node block.
     3. compute meI as  meI := scanvalue + meI - local conN
  */
  int i;
  plural char* old_cons = cons;
  plural _bint lxN = `nd_D.lxN,
               lyN = `nd_D.lyN;
  plural _sint blocksize = _S(lxN + lyN);
  int          con_ls = _sgl(interf_D->con_ls);
  plural _sint local_conN, /* number of connections in each PE */
               sum;        /* prefix sum over local_conN */
  plural _sint myI;
  _TRACE (4, ("update_conI_conN (%x, ls=%d)\n", (int)cons, con_ls));
  if (`no update needed)
    return;
  `compute local conN;
  `compute prefix sum over local conN;
  `compute conI;
  `send sum to rest of node block;
  interf_D->conN = sum;

`no update needed:
   /* must be singular value! */
   !globalor (`nd_D.exists && interf_D->conN == invalid_conN)

`compute local conN:
   for (i = con_ls, local_conN = 0; i > 0; i--, cons += con_size)
     if (`me_D.exists)
       `me_D.meI = local_conN++;
   cons = old_cons;

`compute prefix sum over local conN:
   plural _sint your_xPE, your_yPE, yourPE;
   plural _bint x0 = (plural _bint)ixproc & (plural _bint)~_M(lxN);
   _sint        step = 1;
   _bint        lstep = 0;
   myI = (ixproc & _M(lxN)) + ((iyproc & _M(lyN)) << lxN);
   sum = local_conN;
   while (myI + step < blocksize) {
     your_xPE = x0 + ((ixproc+step) & _M(lxN));
     your_yPE = iyproc + ((ixproc-x0 + step) >> lxN);
     yourPE = your_xPE + (your_yPE << lxprocN);
     sum += router[yourPE].sum;
     step <<= 1;
     lstep++;
   }
```

```
'compute conI:
   plural _sint help = sum - local_conN;
   /* because my local cons are now added individually: */
   for (i = 0; i < con_ls; i++, cons += con_size)
     if ('me_D.exists)
       'me_D.meI += help;

'send sum to rest of node block:
   ss_xsendc (lyN, lxN, (plural void*)&sum, sizeof(sum), myI == 0);

'me_D:
   *(plural _connection_D*)(cons+descr_offset)

'nd_D:
   *(plural _node_D* plural)interf_D->boss

}


_work wpc_sum (plural _interface_D *interf_D)
{
  /* This procedure is called for one interface in one node
     virtualization layer; it computes the sum of the wpc values
     of the interface at the existing nodes.
  */
  plural _work wpcs = 0;
  if (interf_D->boss->exists == _existing)  /* exclude shadows */
    router[interf_D->boss->meI].wpcs = interf_D->wpc;
  return (reduceAdd64u (wpcs));
}
}
```

This macro is defined in definitions 373 and 374.
This macro is invoked in definition 392.

53.4 Machine control and analysis

This module contains operations that either control or analyze the behavior of the machine.

There is a function `pick_PE` that returns the number of a random one of the PEs on which a condition is true.

The function `dpuTimerTicks2` implements reliable DPU timing. The MasPar's ACU has a design error that makes the original `dpuTimerTicks` routine (which measures time in system clock ticks, 80ns each) output wrong values occasionally. I have made a few experiments on our MP-1216A to find a pattern in these faults in order to devise a way to recognize them. These experiments suggest that the following describes the error behavior of the timer values:

Errors occur randomly distributed and happen once in about 12000 (with much variation) `dpuTimerTicks` calls on the average, independent of how much work the program does in between the calls. When a wrong value was returned by `dpuTimerTicks`, succeeding values are still correct in respect to the *same* call of `dpuTimerStart`, i.e., the timer does not have to be restarted in order to recover from an error. Within each single time slice of the DPU process, only two different wrong values are possible; their difference is always 2. Within the next time slice, two other wrong values occur. The new wrong values are smaller than the previous ones, but the difference from the previous wrong seems to depend on the time between the time slices somehow. With just two active DPU processes it is usually in the range of several millions. Two `dpuTimerTicks` calls that are executed immediately after one another, usually return values that differ by about 56 ticks. From time to time, larger differences occur but these are always less than 500.

These observations lead to the following method for reliable timing: Repeat getting a close pair of `dpuTimerTicks` values as long as the difference of the two values is less than 40 or larger than 500 (thresholds should roughly be the same on a MP-2). The resulting two values are then correct with roughly 500 ticks (40 microseconds) precision.

The chance that both values of any pair are wrong is about 12000^2 (144 million), but even this case will usually be caught and repaired since these two wrong values differ only by 2 (if the corresponding observation is always true). Given this situation, the only way to get a wrong value out of the method is that (1) both values of the pair are wrong, (2) between the two calls to `dpuTimerTicks` the timeslice ends, and (3) the old and new wrong values differ by at most 500. The probability of (3) is according to my observations less than 1/160000. This estimation is probably quite conservative: it assumes that the downward steps are distributed evenly between 0 and 80 million, which is true when there are two constantly active DPU processes in DPU memory; with swapping or other long pauses between time slices, steps larger than 80 million occur and the probability is still smaller. Assuming that a program spends as much as 10 percent of its time calling `dpuTimerTicks` and each timeslice is one second long, (2) will roughly happen once every 200 seconds. If the conditions are independent, (1) *and* (2) *and* (3) will then happen once about every 146 million CPU years on the average.

I believe we can live with that.

`dpuTimerTicks2` is used exactly like `dpuTimerTicks`, the only observable differences are that (1) the new operation is slower (it takes roughly 370 ticks) and (2) no bogus values occur. Just like for `dpuTimerTicks`, measurements are returned as 32-bit integer; overflow occurs after 2^{32} ticks, which is 343.6 million microseconds or 5 minutes and 43 seconds. The remaining problem with this routine is that a swapout/swapin cycle may still be visible as consumed time; typical values on our machine are about 400000 to 500000 ticks for a job requiring 4 kB PMEM, but I have also observed outliers with 600000 and 1.9 million. Note that MPPE slows `dpuTimerTicks` down somehow, so that for `dpuTimerTicks2` to work correctly in MPPE sessions, the tolerance needs to be increased to 1000.

There is an additional function `spendTicks` that performs a waiting loop consuming a given number of ticks. This is used for latency simulation purposes.

375 *rts.h Machine Control*[375] ≡

```
{
 _sint _pick_PE (plural _bool condition);
 void     dpuTimerStart ();
 unsigned dpuTimerTicks2 ();
 void     _spendTicks (register unsigned ticks);
 }
```

This macro is invoked in definition 385.

376 *RTS Machine Control*[376] ≡

```
{
 void     dpuTimerStart();
 unsigned dpuTimerTicks();

 _sint _pick_PE (plural _bool condition)
 {
   if (condition)
     return (_sgl(iproc));
   else
     return (0);  /* emergency solution */
 }

 unsigned dpuTimerTicks2 ()
```

```
{
  register unsigned t1, t2, count = 0;
  do {
    t1 = dpuTimerTicks();
    t2 = dpuTimerTicks();
    count++;
  }
  while (t2-t1 > 1000 /*solo:500, with MPPE:1000*/
         || t2-t1 < 10);
  return (t2);
}

void _spendTicks (register unsigned ticks)
{
  register int end = dpuTimerTicks2() + ticks - 200; /* 200 for overhead */
  while ((int)dpuTimerTicks2() < end)
     ;
}
}
```

This macro is invoked in definition 393.

The following code can be used to test the `dpuTimerTicks2` routine. It should display a line after roughly every 500000 iterations showing a difference of more than 900 ticks (but not very much more) in the fourth column. In addition, one larger difference appears for each swapin/swapout cycle.

test dpuTimerTicks2[377] ≡ 377

```
{
   unsigned n, first, second, old = 0, new;
   dpuTimerStart();
   n=1;
   while (n != 0) {
     n=n+1;
     new = dpuTimerTicks2();
     if (new - old < 10 || new - old > 900) {
       printf ("%7d %12d %12d %12d\n", n, old, new, new-old);
       new = dpuTimerTicks2 ();
     }
     old = new;
   }
}
```

This macro is NEVER invoked.

53.5 Output

For the automatic generation of output procedures for all network, node, and connection types, we need basic output procedures for the builtin types and the descriptor types. These are defined here.

In addition we define output procedures for the basic types that can be used in user programs.

rts.h Output Operations[378] ≡ 378

```
{
 void p_pr_network_D (String name, plural _network_D* plural ME);
 void p_pr_node_group_D (String name, plural _node_group_D* plural ME);
 void p_pr_node_D (String name, plural _node_D* plural ME);
 void p_pr_interface_D (String name, plural _interface_D* plural ME);
 void p_pr_connection_D (String name, plural _connection_D* plural ME);
```

```
void p_pr_Gptr (String name, plural _Gptr* plural ME);
void p_prBool (String name, plural Bool* plural ME);
void p_prReal (String name, plural Real* plural ME);
void p_prString (String name, plural String* plural ME);
void p_prInt (String name, plural Int* plural ME);
void p_prInt1 (String name, plural Int1* plural ME);
void p_prInt2 (String name, plural Int2* plural ME);
void p_prRealerval (String name, plural Realerval* plural ME);
void p_prInterval (String name, plural Interval* plural ME);
void p_prInterval1 (String name, plural Interval1* plural ME);
void p_prInterval2 (String name, plural Interval2* plural ME);
void p_pr_remote_connection (String name, plural _remote_connection* plural ME);
void p_pr_remote_connection_interface (String name,
      plural _remote_connection* plural ME, plural _interface_D* plural descr);
}
```

This macro is invoked in definition 385.

379 *RTS Output Operations*[379] ≡

```
{
void p_pr_network_D (String name, plural _network_D* plural ME)
{
  printf ("%s@%x=(", name, (int)_sgl(ME));
  p_printf ("exists=%d formA=%d meI=%d repN=%d lrepN=%d lxN=%d lyN=%d) ",
            (plural int)ME->exists, (plural int)ME->formA, (plural int)ME->meI,
            (plural int)ME->repN, (plural int)ME->lrepN,
            (plural int)ME->lxN, (plural int)ME->lyN);
}

void p_pr_node_group_D (String name, plural _node_group_D* plural ME)
{
  printf ("%s@%x=(", name, (int)_sgl(ME));
  p_printf ("nodesN=%d newnodesN=%d localsizeN=%d better2virt=%d boss=%x) ",
            (plural int)ME->nodesN, (plural int)ME->newnodesN,
            (plural int)ME->localsizeN, (plural int)ME->better2virt,
            (plural int)ME->boss);
}

void p_pr_node_D (String name, plural _node_D* plural ME)
{
  printf ("%s@%x=(", name, (int)_sgl(ME));
  p_printf ("exists=%d meI=%d lxN=%d lyN=%d boss=%x) ",
            (plural int)ME->exists, (plural int)ME->meI,
            (plural int)ME->lxN, (plural int)ME->lyN,
            (plural int)ME->boss);
}

void p_pr_interface_D (String name, plural _interface_D* plural ME)
{
  printf ("%s@%x=(", name, (int)_sgl(ME));
  p_printf ("con_ls=%d boss=%x conN=%d work_per_con=%d wpc=%d) ",
            (plural int)ME->con_ls, (plural int)ME->boss,
            (plural int)ME->conN, (plural int)ME->work_per_con,
            (plural int)ME->wpc);
}

void p_pr_connection_D (String name, plural _connection_D* plural ME)
```

```
{
  printf ("%s@%x=(", name, (int)_sgl(ME));
  p_printf ("exists=%d meI=%d boss=%x) ",
            (plural int)ME->exists, (plural int)ME->meI,
            (plural int)ME->boss);
}

void p_pr_Gptr (String name, plural _Gptr* plural ME)
{
  printf ("%s=(", name);
  p_printf ("pe=%d a=%x) ", (plural int)ME->pe, (plural int)ME->a);
}

void p_prBool (String name, plural Bool* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%c ", (plural int)(*ME ? 'T' : 'F'));
}

void p_prReal (String name, plural Real* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%g ", *ME);
}

void p_prString (String name, plural String* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("`%s' ", *ME);
}

void p_prInt (String name, plural Int* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%d ", *ME);
}

void p_prInt1 (String name, plural Int1* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%d ", (plural int)*ME);
}

void p_prInt2 (String name, plural Int2* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%d ", (plural int)*ME);
}
```

```
void p_prRealerval (String name, plural Realerval* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%g...%g ", ME->min, ME->max);
}

void p_prInterval (String name, plural Interval* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%d...%d ", ME->min, ME->max);
}

void p_prInterval1 (String name, plural Interval1* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%d...%d ", (plural int)ME->min, (plural int)ME->max);
}

void p_prInterval2 (String name, plural Interval2* plural ME)
{
  if (*name != 0)
    printf ("%s=", name);
  p_printf ("%d...%d ", (plural int)ME->min, (plural int)ME->max);
}

void p_pr_remote_connection (String name, plural _remote_connection* plural ME)
{
  printf ("\n%s@%x=(", name, (int)_sgl(ME));
  p_pr_connection_D ("_me_D", &ME->_me_D);
  p_pr_Gptr ("_oe", &ME->_oe);
  printf (")");
}

void p_pr_remote_connection_interface (String name,
      plural _remote_connection* plural ME, plural _interface_D* plural descr)
{
  char* n = "0";
  int  i;
  printf ("\n%s@%x=(", name, (int)_sgl(ME));
  p_pr_interface_D ("_me_D", descr);
  for (i = 0; i < descr->con_ls; i++, ME++) {
    *n = (i % 64) + '0';  /* name the individual connections consecutively */
    p_pr_remote_connection (n, ME);
  }
  printf (")");
}
}
```

This macro is invoked in definition 394.

54 Miscellany

This section comprises those parts of the run time system that are not directly used in the generated program. These are the front end program (which must be linked with the generated program) plus some administrative infrastructure: A compiler driver shell script and two makefiles.

54.1 The front end program

The code generated by the CuPit compiler runs on the ACU and the PE array only. But it nevertheless has to be started on the front end machine. The following is a little C program to be run on the front end in order to parse command line arguments and start the CuPit program. A command line argument preceeded by an argument consisting of only the letter 'T', if present, is used as an integer in order to set the `tracelevel` variable of the program. Allowed values are from 0 (no tracing) to 5 (maximum tracing). A command line argument preceeded by an argument consisting of only the letter 'R', if present, is used as the seed for the random number generator All following arguments are treated as either floating point values or strings: If an argument appears that consists of only the letter 'N', this argument is skipped and the next argument is interpreted as a string argument and copied to the `names` variable. Otherwise, arguments are interpreted as floating point arguments and their values are copied to the `args` variable (which must be an array of at least 20 floats). The number of such floating point arguments is copied to the `argsN` variable; the number of string arguments is copied to the `namesN` variable. Afterwards, the driver calls the initialization procedure `INIT` and then the main procedure `program`. The command line argument descriptors `argc` and `argv` are made available as global variables to be fetched by `copyIn` by the MPL program if necessary.

lib/callCupit.c[380] ≡ 380

```
{/*
 File: CuPit front end program for MasPar
 RCS:  $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */

 #include <stdio.h>
 #include <stdlib.h>

 extern INIT ();
 extern program ();
 extern print_rusage ();
 extern int _tracelevel;
 extern int _randominit;
 extern float _args[];
 extern int   _argsN;
 extern int   _nameoffsets[];
 extern int   _namesN;
 extern char  _names[];

 int    argc;
 char **argv;
 static float floatargs[20];
 static int   nameoffsets[10];
 static char  names[210];

 int main (int _argc, char *_argv[])
 {
   int trclv,
       rndinit,
       i = 1,
```

```
        floatargsN = 0,  /* number of float arguments already found */
        stringargsN = 0, /* dito for string arguments */
        namespos = 0;    /* sum of lengths of string arguments so far */
    argc = _argc;
    argv = _argv;
    if (argc < 2) {
      fprintf (stderr,
          "usage: cmd [T tracelevel] [R randominit] [N namearg | floatarg]*\n");
      return (1);
    }
    /***** parse command line arguments: */
    while (i < argc) {
      if (argv[i][0] == 'T') {
        i++;
        trclv = atoi (argv[i]);
        copyOut (&trclv, &_tracelevel, sizeof (int));
        printf ("----- StartFE\n");
      }
      else if (argv[i][0] == 'R') {
        i++;
        rndinit = atoi (argv[i]);
        copyOut (&rndinit, &_randominit, sizeof (int));
      }
      else if (argv[i][0] == 'N') {
        i++;
        nameoffsets[stringargsN++] = namespos;
        strcpy (names+namespos, argv[i]);
        namespos += strlen (argv[i]) + 1;
      }
      else {
        floatargs[floatargsN++] = atof (argv[i]);
      }
      i++;
    }
    copyOut (&floatargsN, &_argsN, sizeof (int));
    copyOut (floatargs, _args, sizeof (floatargs));
    copyOut (&stringargsN, &_namesN, sizeof (int));
    copyOut (nameoffsets, _nameoffsets, sizeof (nameoffsets));
    copyOut (names, _names, sizeof (names));
    /***** now execute the DPU program: */
    callRequest(INIT, 0);
    callRequest(program, 0);
    callRequest(print_rusage, 0);
    /***** finish: */
    if (trclv > 0)
      printf ("----- EndFE\n");
    return (0);
  }
  }
```

This macro is attached to an output file.

54.2 Compiler driver script

Since the compiler works in several phases (which even run on different machines), a script is needed that drives the compilation from one phase to the other. This script is simply called `cupit` because it is the program the compiler user will call.

cupit[381] ≡ 381

```
{#!/bin/sh
 # RCS: $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 # File:  CuPit compiler driver script
 # Usage: see message below

 set -e    #exit on error
 set -u    #unset variables are errors

 #----- programs and variables:
 CUPITDIR=${CUPITDIR:-"/home2/i41s25/prechelt/cupit2"}  #set if not predefined
 LIBDIR=$CUPITDIR/lib
 cupit=$CUPITDIR/cc/cupit.exe
 crefine=crefine
 mpl="mpl_cc -Zq"
 cpp="/lib/cpp -D_parallel_=1 -C"

 #----- variables:
 cppopts=""
 cupitopts=""
 mplopts=""
 unknownopts=""
 cupitfiles=""
 mplfiles=""
 ofiles=""
 outfile="a.out"
 unknownfiles=""

 compileonly=0
 check=0
 codetype=0
 nomplcc=0
 pretty=0
 pmem=0
 pe=0
 P=0


 #----- check usage:
 if test $# -eq 0 ; then
   echo "Usage: cupit [options] (mycupit.nn|mympl.m|mympl.mr|myobj.o)...
    Expects only one .nn file but arbitrarily many .m, .mr, and .o
    files. Leaves cupit-compiled version of .nn file in nn1.m, preprocessed
    nn1.m in nn2.m, and line-numbering-stripped nn2.m in nn3.m
    Options:
      -c       compile only, do not link
      -check   exit after syntax check of first '.nn' file (nn1.m is produced)
      -conatout   place connection objects at OUT interfaces (default: at IN)
      -conindividual  fetch/send elements of remote connections individually
      -conwhole   always fetch/send the whole connection object
      -dumbbalance    load balancing assumes connection fetch/send costs nothing
      -Dn=b    define macro n with body b for /lib/cpp
      -hidelatency    make measurement ignore remote connection send/fetch times
      -highlatency    simulate additional latency on remote con send/fetch
```

```
      -nobalance  use regular instead of load-balancing data distribution
      -nodatalocality perform remote fetch/send even for local connections
      -nomplcc generate MPL code but don't compile it to .o files
      -nonodevirt use 1 nodevirt layer where 2 were better and vice versa
      -o file  file to write executable to
      -O       turn on optimizations of MPL compiler (turn off profiling)
      -opt     optimize CuPit, using run time information
      -pretty  run nn3.m through beautifier: gindent -kr
      -pmem 8k sets PMEM limit to 8kb (default:4kb), also possible: 12k, 16k etc
      -pe      use only 1024 PEs (useful for debugging purposes)
      -P       produce print procedures
      -rti     generation run time information collection code
   "
   exit 1
fi

#----- here we go:
while test $# -gt 0 ; do
    case $1 in
       -D*)     cppopts="$cppopts $1" ;;
       -P)      cupitopts="$cupitopts $1"; P=1 ;;
       -g)      mplopts="$mplopts $1" ;;
       -c)      compileonly=1 ;;
       -check)  check=1 ;;
       -conatout)       cupitopts="$cupitopts $1" ;;
       -conindividual)  cupitopts="$cupitopts $1" ;;
       -conwhole)       cupitopts="$cupitopts $1" ;;
       -dumbbalance)    cupitopts="$cupitopts $1" ;;
       -hidelatency)    cupitopts="$cupitopts $1" ;;
       -highlatency)    cupitopts="$cupitopts $1" ;;
       -nobalance)      cupitopts="$cupitopts $1" ;;
       -nodatalocality) cupitopts="$cupitopts $1" ;;
       -nomplcc)        nomplcc=1 ;;
       -wrongnodevirt)  cupitopts="$cupitopts $1" ;;
       -o)      outfile=$2; shift ;;
       -opt)    codetype=2 ;;
       -O)      mplopts="$mplopts -nohprofile" ;; #-Zn == (not -g)
       -pretty) pretty=1 ;;
       -pmem)   pmem=$2; shift ;;
       -pe)     pe=1 ;;
       -rti)    codetype=1 ;;
       -*)      unknownopts="$unknownopts $1" ;;
       *.nn)    echo "$cpp -I"$LIBDIR" $cppopts $1 >nn1.nn"
                $cpp -I"$LIBDIR" $cppopts $1 >nn1.nn  #pass comments
                echo "$cupit $cupitopts -codetype $codetype nn1.nn >nn1.m"
                $cupit $cupitopts -codetype $codetype nn1.nn >nn1.m
                if test $check -eq 1 ; then
                  exit;
                fi
                echo "$mpl -I"$LIBDIR" -E nn1.m >nn2.m"
                $mpl -I"$LIBDIR" -E nn1.m >nn2.m
                echo "removing #line  commands in nn2.m  >nn3.m"
                perl -ne 's/[ ]*$//; print if(!m/^# [0-9]/ && ($_ ne "\n"
                  || $l ne "\n")); $l=$_;' nn2.m >nn3.m
                if test $pretty -eq 1 ; then
```

```
                echo mv -f nn3.m nn3.ugly
                mv -f nn3.m nn3.ugly
                #old: cb -js -l 77 nn3.ugly | expand -3 >nn3.m
                echo  "gindent -kr -st nn3.ugly | perl..."
                gindent -kr -st nn3.ugly |
                  perl -pe 's/^} plural/}\n\nplural/' >nn3.m
              fi
              if test -r dump.m -a $P -eq 1; then  #debugging aid
                echo "cat dump.m >> nn3.m"
                cat dump.m >> nn3.m
              fi
              if test $nomplcc -eq 0; then
                echo $mpl $mplopts -c -nocpp nn3.m
                $mpl $mplopts -c -nocpp nn3.m
                ofiles="$ofiles nn3.o"
              fi
              ;;
      *.mr)   echo $crefine $1
              $crefine $1
              echo $mpl $cppopts $mplopts -I"$LIBDIR" -c `basename $1 r`
              $mpl $cppopts $mplopts -I"$LIBDIR" -c `basename $1 r`
              ofiles="$ofiles `basename $1 .mr`.o" ;;
      *.m)    echo $mpl $cppopts $mplopts -I"$LIBDIR" -c $1
              $mpl $cppopts $mplopts -I"$LIBDIR" -c $1
              ofiles="$ofiles `basename $1 .m`.o" ;;
      *.o)    ofiles="$ofiles $1" ;;
      *)      unknownfiles="$unknownfiles $1" ;;
    esac
    shift
  done
  if test "$unknownopts" != "" ; then
    echo "Unknown option(s) '$unknownopts' ignored"
  fi
  if test "$unknownfiles" != "" ; then
    echo "File(s) '$unknownfiles' with unknown extensions ignored"
  fi
  if test $compileonly -eq 0 ; then
    echo $mpl $mplopts -o $outfile $ofiles $LIBDIR/callCupit.o -L"$LIBDIR" -lcupit
    $mpl $mplopts -o $outfile $ofiles $LIBDIR/callCupit.o -L"$LIBDIR" -lcupit
    if test $pmem != "0" ; then
      mplimit -Zq $outfile pmem $pmem
      mplimit -Zq $outfile
    fi
    if test $pe -ne 0 ; then
      mpswopt -1 $outfile   #1024 PEs only for debugging
    fi
  fi
  }
```

This macro is attached to an output file.

54.3 The compiler Makefile

To extract this Makefile from the FunnelWeb document after changes, use `make makefile` (which also tangles all files belonging to the run time system).

Since FunnelWeb does not like unprintable characters inside macro bodies, we use the following macro to represent the Tab character required by `make` at the beginning of each action line:

382 T[382] $\equiv$

{ }

This macro is invoked in definitions 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 383, 384, 384, 384, 384, 384, 384, 384, 384, 384, 384, 384, 384, and 384.

383 **Makefile**[383] $\equiv$

```
{#Makefile for miscellaneous CuPit compiler administration tasks
#Lutz Prechelt, 93/09/14
#$Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $

CI=ci
CIOPTS= -diff -l
TANGLE= fw +D -L +S1 +U
SCALL=

std:
T[382] @echo "you can make: makefile, lib, ci, tex, dvi, dvi2, clean, or exe"

makefile: lib/.rts.fw_tangled

ci:
T[382] $(CI) $(CIOPTS) compiler.fw
T[382] $(CI) $(CIOPTS) scanner.fw
T[382] $(CI) $(CIOPTS) grammar.fw
T[382] $(CI) $(CIOPTS) names.fw
T[382] $(CI) $(CIOPTS) types.fw
T[382] $(CI) $(CIOPTS) usage.fw
T[382] $(CI) $(CIOPTS) code.tex
T[382] $(CI) $(CIOPTS) code1.fw
T[382] $(CI) $(CIOPTS) code2.fw
T[382] $(CI) $(CIOPTS) rts.fw
T[382] $(CI) $(CIOPTS) aux.fw
T[382] $(CI) $(CIOPTS) appendix.tex
T[382] $(CI) $(CIOPTS) cupit.specs
T[382] $(CI) $(CIOPTS) README
T[382] $(CI) $(CIOPTS) fileinout.fw

tex:
T[382] fw +T -O +S1 -L compiler.fw

dvi: tex
T[382] latex compiler.tex

dvi2:
T[382] #bibtex compiler
T[382] makeindex -o compiler_idx.tex -s $(HOME)/lib/makeindexstyle compiler
T[382] latex compiler.tex
T[382] latex compiler.tex

lib:  lib/.code1.fw_tangled  lib/.rts.fw_tangled  lib/.fileinout.fw_tangled
T[382] @#$(SCALL) cd ../lib
T[382] -cd lib; make
```

T[382] @#$(SCALL) cd ../cc

lib/.code1.fw_tangled: code1.fw
T[382] $(TANGLE) code1.fw
T[382] @rm code1.map
T[382] @touch lib/.code1.fw_tangled

lib/.rts.fw_tangled: rts.fw
T[382] $(TANGLE) rts.fw
T[382] @rm rts.map
T[382] @chmod +x cupit # mode is not preserved by FunnelWeb
T[382] @touch lib/.rts.fw_tangled

lib/.fileinout.fw_tangled: fileinout.fw
T[382] $(TANGLE) fileinout.fw
T[382] @rm fileinout.map
T[382] @touch lib/.fileinout.fw_tangled

clean:
T[382] cd lib; make clean
T[382] rm -f *~ *.lis *.jrn *.map *.log *.o ?

exe:
T[382] eli <F
}

This macro is attached to an output file.

54.4 The library Makefile

The library Makefile is responsible for converting the `.tplr` template files into `.tpl` files by sending them through the C-Refine preprocessor and for compiling the modules of the run time system. For convenience, this makefile also builds (and includes into the library file) some modules that are not directly part of the run time system. Currently the only such module is **`fileinout`**. The Makefile is usually started from the compiler Makefile without an argument. In order for the compilation to work, the **`sexec`** process must have been started on the MasPar (in the correct directory!) before.

To extract this Makefile from the FunnelWeb document after changes, use **`make makefile`** (which also tangles all files belonging to the run time system).

lib/Makefile[384] $\equiv$ 384

{#Makefile for CuPit compiler 'lib' subdirectory management
#Lutz Prechelt, 93/09/14
#$Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $

CR=crefine
SCALL=
MPL= $(SCALL) mpl_cc
MPLFLAGS= -Zq

%.tpl: %.tplr
T[382] @echo $< :
T[382] @rm -f $*.tpl
T[382] @$(CR) -n0 -s- -c $*.tplr
T[382] @chmod -w $*.tpl

```
%.o: %.mr
T[382] @echo $< :
T[382] @$(CR) -c -n0 -s- $*.mr
T[382] @$(MPL) $(MPLFLAGS) -c $*.m

tpls= \
NodeArrayInit.tpl        ArrayInit.tpl \
ReductionCon.tpl         ReductionNode.tpl        ReductionNet.tpl \
WtaCon.tpl               WtaNode.tpl              WtaNet.tpl \
Input.tpl                Output.tpl \
Connect.tpl \
MergeCon.tpl             MergeNode.tpl            MergeNet.tpl \
Extend.tpl \
ReplicateNode.tpl        ReplicateNet.tpl         ReplicateNetNodes.tpl

objs= \
rtsrandom.o \
rtsoperators.o \
rtstopology.o \
rtsstdlib.o \
rtscomm.o \
rtsmalloc.o \
rtsaddress.o \
rtsmachine.o \
rtsoutput.o \
fileinout.o

#And this is what must be done:

lib: templates objects libcupit.a callCupit.o

templates: $(tpls)

objects: $(objs)

libcupit.a: $(objs)
T[382] @#scall "mpar -cr libcupit.a \`mplorder" $(objs) "| tsort\`" #use this
T[382] mpar -cr libcupit.a `mplorder $(objs) | tsort`             # OR this
T[382] $(SCALL) mpranlib -Zq libcupit.a

callCupit.o: callCupit.c
T[382] @echo -n "in maspar:"; scall pwd
T[382] scall $(MPL) $(MPLFLAGS) -c $*.c

clean:
T[382] rm -f *~ *.lis *.jrn *.map *.log *.o ?

}
```

This macro is attached to an output file.

55 Put it all together

Here we construct the files comprising the run time system from the parts given above. There is one header file `rts.h` that defines the whole run time system plus one MPL file for each of the parts random numbers, operators, topology change, communication, memory allocation, and computation.

lib/rts.h[385] ≡ 385

```
{/*
 File:   CuPit run time system header file
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #ifndef RTS_H
 #define RTS_H
 #include <mpl.h>
 #include <maspar/mp_libc.h>
 #include <mp_resource.h>  /* for mpGetRUsage */
 #undef NULL  /* else we get 'redefined' message from stdio.h */
 #include <stdio.h>
 #include <stddef.h>
 #include <stdlib.h>
 #include <string.h>
 #include "mplforgotten.h"
 #include "cupittypes.h"
 #include "descriptors.h"
 #include "libmisc.h"

 rts.h Random Number Generator[340]
 rts.h Type Conversions[342]
 rts.h Operators[343]
 rts.h Topology Change Operations[345]
 rts.h Standard Library[353]
 rts.h Communication Functions[368]
 rts.h Memory Allocation[370]
 rts.h Address Computations[372]
 rts.h Machine Control[375]
 rts.h Output Operations[378]

 #endif
}
```

This macro is attached to an output file.

(Just like any other section of the same name, most of this section is pretty booooring:)

lib/rtsrandom.mr[386] ≡ 386

```
{/*
 File:   CuPit run time system random number generator
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 RTS Random Number Generator[341]
}
```

This macro is attached to an output file.

lib/rtsoperators.mr[387] ≡ 387

```
{/*
 File:   CuPit run time system interval operations and other operators
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 RTS Operators[344]
```

}

This macro is attached to an output file.

388 **lib/rtstopology.mr**[388] ≡

```
{/*
 File:   CuPit run time system topology change operations
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 RTS Topology Change Operations[346]
 }
```

This macro is attached to an output file.

389 **lib/rtsstdlib.mr**[389] ≡

```
{/*
 File:   CuPit run time system standard library
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 #include <math.h>
 #include <mp_libm.h>
 /* #include <mp_time.h> gives "redefinition of ‘struct timeval’" error */
 RTS Standard Library[354]
 }
```

This macro is attached to an output file.

390 **lib/rtscomm.mr**[390] ≡

```
{/*
 File:   CuPit run time system communication operations
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 RTS Communication Operations[369]
 }
```

This macro is attached to an output file.

391 **lib/rtsmalloc.mr**[391] ≡

```
{/*
 File:   CuPit run time system memory allocation
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 RTS Memory Allocation[371]
 }
```

This macro is attached to an output file.

392 **lib/rtsaddress.mr**[392] ≡

```
{/*
 File:   CuPit run time system address computations
 RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
 */
 #include "rts.h"
 RTS Address Computations[373]
 }
```

This macro is attached to an output file.

393 **lib/rtsmachine.mr**[393] ≡

```
{/*
File:   CuPit run time system machine control and analysis operations
RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
*/
#include "rts.h"
RTS Machine Control[376]
}
```

This macro is attached to an output file.

lib/rtsoutput.mr[394] ≡ 394

```
{/*
File:   CuPit run time system output operations
RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
*/
#include "rts.h"
RTS Output Operations[379]
}
```

This macro is attached to an output file.

lib/stdlib.nn[395] ≡ 395

```
{
/*
File:   CuPit standard library Cupit declarations
RCS:    $Id: rts.fw,v 1.16 1994/11/07 10:58:16 prechelt Exp prechelt $
*/
stdlib.nn Output Procedures[355]
stdlib.nn Arithmetic Functions[361]
stdlib.nn Other Procedures[364]
}
```

This macro is attached to an output file.

PART VI: Auxiliary Stuff

The *auxiliary stuff* are several small files for various purposes that do not fit well into any other place.

396 **aux.head**[396] ≡

{
LIDO Extensions[397]
Option Variables[400]
}

This macro is attached to an output file.

56 LIDO extensions

We define an additional head file for Lido that defines macros for printing messages, computing the number of error messages emitted, and converting definition table keys into PTG nodes directly.

397 *LIDO Extensions*[397] ≡

{
```
/* $Id: aux.fw,v 1.10 1994/11/07 10:58:53 prechelt Exp prechelt $ */
/* messages with several additional parameters: */
#define Message(s, m)   message (s, m, 0, COORDREF)
#define Message1(s, m, p1)          \
  message (s, sprintf (malloc(100), m, p1), 0, COORDREF)
#define Message2(s, m, p1, p2)      \
  message (s, sprintf (malloc(120), m, p1, p2), 0, COORDREF)
#define Message3(s, m, p1, p2, p3)  \
  message (s, sprintf (malloc(140), m, p1, p2, p3), 0, COORDREF)
#define Message4(s, m, p1, p2, p3, p4)  \
  message (s, sprintf (malloc(160), m, p1, p2, p3, p4), 0, COORDREF)

/* pseudo-macros for currently inactivated messages: */
#define Messag(s, m)                4710
#define Messag1(s, m, p1)           4711
#define Messag2(s, m, p1, p2)       4712
#define Messag3(s, m, p1, p2, p3)   4713
#define Messag4(s, m, p1, p2, p3, p4) 4714

/* the number of errors that occured during compilation: */
#define NrOfErrors  (ErrorCount[ERROR] + ErrorCount[FATAL])

/* Make a PTG node from the symbol stored under a DefTableKey */
#define PTGKey(k)  PTGStr (SymString (GetSym (k, NoSym)))

/* Prepend a comma to a PTGNode if it is nonempty (for argument lists) */
#define WithComma(p) ((p)==PTGNULL ? (p) : PTGSeq (PTGStr (", "), (p)))

/* Bit operations on integers: */
#define BITAND(a,b)  ((a) & (b))
#define BITOR(a,b)   ((a) | (b))
#define BITNOT(a,b)  ~(a)
```
}

This macro is invoked in definition 396.

57 cupit.h

The file cupit.h defines the basic types of CuPit for use in data objects of the compiler itself.

cupit.h[398] ≡ 398

```
{
 #ifndef cupit_H
 #define cupit_H
 /* $Id: aux.fw,v 1.10 1994/11/07 10:58:53 prechelt Exp prechelt $ */
 /* for use internally in the compiler only */

 #include <stdio.h>    /* for fprintf() in _assert() */
 #include <stdlib.h>   /* for atof() */
 #include "csm.h"      /* string[] */

 typedef char*  String;
 typedef int    Bool;
 typedef int    Int;
 typedef short  Int2;
 typedef char   Int1;
 typedef double Real;

 #define true   1
 #define false  0
 #define BoolNull() 0    /* for CONSTITUENTS */

 /* Turning a .Sym attribute value into a string: */
 #define SymString(s)  ((s) < 0 ? "<unknown>" : string[s])
 #define NoSym         -1

 /* assertions: */
 #ifndef NDEBUG
 #define _assert(c) ((c) ? 0 : fprintf (stderr, "Oops! '%s', line %d\n", \
                                         __FILE__,  __LINE__))
 #else
 #define _assert(c)
 #endif

 #endif
}
```

This macro is attached to an output file.

58 Command line processing

The following declarations introduce options that are automatically processed by the generated compiler. Each boolean option is available as an integer variable, each integer or string option or positional argument is available as a definition table key (the corresponding integer or string value is SymString(GetValue(x))).

options.clp[399] ≡ 399

```
{
 /* $Id: aux.fw,v 1.10 1994/11/07 10:58:53 prechelt Exp prechelt $ */
 /*---------- Control optimizations: */
 conAtOut                "-conatout"  boolean
```

```
    "Locate connection data at OUT interfaces (default: IN interfaces)";
 conIndividual          "-conindividual"  boolean
    "Fetch/send used elements of (remote) connection objects individually";
 conWhole               "-conwhole"  boolean
    "Always fetch and send the whole (remote) connection object";
 dumbBalance            "-dumbbalance"  boolean
    "Assume for load balancing that remote connection fetch/send costs nothing";
 noBalance              "-nobalance"  boolean
    "Use regular instead of load-balancing data distribution";
 noDataLocality         "-nodatalocality" boolean
    "Generate dummy send/fetch for local connection operations";
 hideLatency            "-hidelatency" boolean
    "Make timing functions ignore remote connection fetch/send times";
 highLatency            "-highlatency" boolean
    "Add generate code with artificial additional latency";
 wrongNodeVirt          "-nonodevirt"  boolean
    "Use one node virtualization layer where two were better and vice versa";
 codetype               "-codetype"  int
    "0 = plain   1 = collect run time inforation   2 = optimized";
 /*---------- other: */
 ProducePrintcode       "-P"  boolean
    "Produce print procedures";
 FileName               positional
    "File name of CuPit program";
 }
```

This macro is attached to an output file.

400 *Option Variables*[400] $\equiv$

```
{
 #include "clp.h"
 }
```

This macro is invoked in definition 396.

APPENDIX

A cupit.specs

This section contains the file `cupit.specs` which describes to Eli the set of files that together form the specification of a compiler. In our case we find there a number of FunnelWeb files that contain the individual parts of the hand-written compiler specification plus a number of instantiations of Eli library modules:

```
/*
File: ELI .specs file for MasPar CuPit compiler specifications
RCS:  $Id: cupit.specs,v 1.7 1994/02/28 12:12:49 prechelt Exp prechelt $
*/

scanner.fw
cpp.fw
grammar.fw
names.fw
types.fw
usage.fw
code2.fw
aux.fw

LIGA.ctl

$/Tool/lib/Name/Chain.gnrc :inst
$/Tool/lib/Name/NoKeyMsg.gnrc :inst
$/Tool/lib/Name/Unique.gnrc :inst
$/Tool/lib/Name/Field.gnrc :inst

$/Tool/lib/Tech/GenName.gnrc +referto=_tmp :inst
$/Tool/lib/Tech/LeafPtg.gnrc :inst
$/Tool/lib/Tech/Indent.gnrc :inst
$/Tool/lib/Tech/NodCoord.specs

/usr/lib/libm.a
```

B Compiler restrictions

This section describes what restrictions and peculiarities the compiler has compared to the CuPit language definition.

1. The number of replicates of any single network is limited to the number of processors of the target machine.
2. The number of nodes in any single node group is limited to the number of processors of the target machine.
3. No names must be used in a CuPit program that are keywords in MPL.
4. Some of the operators evaluate their operands twice.
5. Applying `REPLICATE ME INTO 0` to nodes that belong to a node array (not a node group) does not produce a run time error. Instead, it works just as if the array was a group.
6. Procedures that are meant to be part of the central agent but formally are not, may induce problems (usually illegal assigments of plural values to singular variables in the unwanted parallel version of the procedure). Artificially mention a network variable to make the procedure part of the central agent to resolve the problem.
7. There are some holes in the semantic checking, in particular concerning illegal VAR parameters.
8. Records are initialized only when they are members of connections, nodes, or networks, but not when they are individual objects.
9. In `CONNECT` statements, no integers can be used for the description of node group slices; intervals of the form `n...n` have to be used instead.

10. When a connection operation `b`, which was called from a connection operation `a`, deletes the connection (`REPLICATE ME INTO 0`), only `b` is immediately returned from, but the rest of `a` after the call to `b` is still executed for the deleted connections — usually without any effect, though. The analog is true for nodes.
11. When the input or output assignment is used for a non-builtin type, the procedures implementing it will appear before the definition of the type itself in the resulting code.
12. Calls using a slice of a network (i.e. only certain replicates) will not compile. Slicing only works properly for groups.
13. The random number generator does not know about node blocks. As a result, using it in node or network procedures will lead to inconsistencies since different values are returned on different processors.
14. Record procedures are not handled properly.
15. Arrays (except for node arrays) are never intialized.
16. Replication of a node into many nodes is not implemented.
17. Connection addressing is not implemented.
18. Assignment of complete objects of structure types and constructors for structure types are not implemented.
19. Reductions and winner-takes-all operations on arrays are not implemented. (Package the array type into a record type to perform a reduction or WTA on it.)
20. The '`%=`'-assignment is not implemented for `Reals`.
21. The compiler assumes that the MPL compiler casts integer arguments passed to float parameters automatically, which it does not.
22. The compiler does not detect when type conversions from type A to B are applied to arrays of A.
23. The compiler does not complain about assignments to arrays; not even for scalars of a type other than the array element type.
24. Comments that extend beyond the end-of-file produce a segmentation fault.

C List of exit codes

When a run time error occurs, the program terminates with an exit code other than 0. Below is a list of standard error codes used by programs generated by the CuPit compiler along with their meaning.

Code	Meaning
10	number of network replicates must be positive
11	`DISCONNECT` not allowed for replicated networks
12	`REPLICATE` connection `INTO` (many) impossible
13	`REPLICATE` connection called while network is replicated
14	`CONNECT` called while network is replicated
15	`EXTEND` not allowed for replicated networks
16	`EXTEND` BY (negative) called, but group has only (fewer) nodes
17	`REPLICATE` node not allowed in replicated networks
18	`REPLICATE` node `INTO` x only allowed for x=0 and x=1
19	`REPLICATE` net called for replicated network
31	not enough memory
32	too many nodes per segment
41	could not write tracefile
42	could not open datafile
43	datafile format error or datafile inconsistency
44	openDatafile: dump was for x PEs, I have (more)
45	could not open or write file to dump to
46	could not write output file

47 could not write protocol file
1 (other)

D I/O area handling

An I/O area object `a` of type `X` that is passed to an external C procedure is passed as a `plural X **a`. External procedures are responsible for allocating enough memory for `*a`. The memory layout of the I/O area is just a plural array of `X` values at a singular start address, where coefficient `c` of replicate `n` is stored at

```
proc[(n*N+c)%P].(*a)[(n*N+c)/P]
```

(where both `c` and `n` are counted beginning at 0; `N` is the number of nodes; `P` is the number of processors).

E Errors during compiler development

This appendix presents an annotated list of all errors found in the compiler during its test and use. For each error, I recorded the type of error according to the taxonomy described below, the date (in German format: DD.MM.) and time when the error was found, and a short description of the error. This list may be useful for anybody who wants to study programmer errors. It was made in the spirit of Knuth's article "The errors of TEX" [Knu89]. The error taxonomy used in my list is a partially simplified and partially extended version of the one used by Knuth. There are the following error types:

A. Algorithm Awry. An error in the algorithmic program logic. This error accounts for 40% of the 89 errors in the list.

C. Incomplete Change. I intended to change some aspect of the program (usually one that was relevant in more than one place), but did not change all relevant parts of the program or changed a particular part only incompletely. 2% of the errors.

D. Data Structure Debacle. The invariants of a data structure were violated, i.e., either never established or not maintained. 20% of the errors.

F. Forgotten Function. Some part of the program simply had never been implemented, although I thought it was. 2% of the errors.

L. Language Liability. An error introduced by peculiarities of the programming language used. Such errors would not have occured in "better" languages. 13% of the errors.

M. Module Mismatch. A function was used in a way that did not satisfy its interface conditions. 7% of the errors.

P. Parallelism Perplexity. Too many, too few, or the wrong PEs were active at some point during program execution. 3% of the errors.

T. Trivial Typo. I didn't type the characters that I intended to type, resulting in a legal but wrong program. 1% of the errors.

X. Mix-up of data objects of functions. Used object A where B would have been correct. The difference to the trivial type is that I more or less *believed* A was correct. 3% of the errors.

1. Off-by-one error. Using an integer number that was too large or too small by one; for instance by writing `<=` instead of `<` in a loop condition. 7% of all errors.

It should be noted that in many cases it is not clear which category an error should be assigned to; thus the above percentages are fuzzy. Below follows the error list. Errors 1 to 31 were found during a code inspection from November 10, 1993, to November 23, 1993. In this inspection, code produced by an early version of the compiler was considered before any program execution was attempted. Subsequent errors were found during actual tests of the compiler-produced code, which occured in parallel to further compiler development (introduction of various optimizations etc). The first version of the compiler that produced a successfully running program was after error 55. The error list was originally written in German language. The list was only translated and typeset, but the contents were not edited; so you can see my state of mind (when I found the error) shine through in various places. I hope I did not introduce too many errors into the errors during the translation...

1. (D) 10.11., 9:38
   Network descriptor is not initialized (`INIT_Mlp`).
2. (C) 10.11., 9:59
   In `compute_block_size0`: `_log2 (_sgl (net_D->lxN))` seems to be leftover from change xN ::= lxN in `_network_D`.
3. (L) 10.11., 12:05
   `xx_xfetch (dy, o, ....)` goes `dy` steps northwards, not southwards. Ditto for `xx_xsend`. This means there are errors in various places in the templates.
4. (A) 10.11., 12:35
   In `a_MERGE_Net`: for y-reduction and redistribution do not use `if (iyproc == 0)` but use `if (iyproc < _S(net->_me_D.lxN)` instead, since we need a complete replicate as the result, not only on PE 0.
5. (D) 10.11., 14:06
   In `REPLICATE_Net` in redistribute-step: `result_computed` set to wrong value. `_me_D.meI == 0` is insufficient, `_me_D.exists` must be true as well.
6. (A) 10.11., 14:13
   In `REPLICATE_Net` in redistribute-steps: Condition for `‘put x value` is too weak. Additionally `‘x_neighborI < repN` must be satisfied, otherwise we may overwrite non-existing replicates and wrongly set `exists` to true there. Analogously for y value.
7. (D) 10.11., 14:26
   In `REPLICATE_Net` we must compute `meI` in `create_replicates`.
8. (D) 10.11., 16:52
   In `REPLICATE_Net` the `repN` of `net_D` is not set for the `_A_to_0` case.
9. (A) 10.11., 17:04
   In `REPLICATE_Net`: upon initialization of `net_D` for cases `_0_to_0` and `_0_to_A` we compute wrong values for `net_D.lxN` and `lyN`.
10. (F) 11.11., 11:06
    In `REPLICATE_Net` at `‘compute work and conN and conI` forgot to transfer `mywork` to `nd->_me_D.work`.
11. (D) 15.11., 8:55
    In `layout__` no values set for `group_D->nodesN` and `group_D->boss`.
12. (D) 15.11., 9:30
    In `layout__` at `‘init _me_D of new _existing nodes` no value set for `nd_D.work`.
13. (1) 15.11., 9:31
    In `layout__` at
    `‘send _existing old node to each new node and allocate connection arrays` the first loop with `0...localsize-1` is too short by one iteration.
14. (A) 15.11., 9:38
    Ditto: `for (i = layersN - 1; i <= 0; i--, nd++)` is a brain-dead loop. In the future we write `for (i = 0; i < layersN; i++, nd++)`. This change also removes another error which would have occured due to the test `if (layer == i)` and the subsequent operations on `nd`, because `i--,nd++` does not guarantee the implicitly assumed `*nd == nodes[i]`.
15. (A) 15.11., 9:45
    Ditto: in `_lfold3` we must not use `lxN,lyN` but `lxblocksize,lyblocksize` (which must be made into variables that are global to the procedure for this purpose).

16. (A) 15.11., 10:05
    The same problem occurs in
    `‘fetch _existing old node to each new node and allocate connection arrays` (In order to have proper values for `lxblocksize,lyblocksize` there, we also set these variables after calling `compute_block_sizesA`). Fehler (14) tritt hier ebenso nochmals auf.
17. (A) 15.11., 10:24
    In `copy_connections` we must dereference when initializing `con`, because we want to know the address of the connections and not the address at which this address is stored.
18. (A) 15.11., 10:28
    In `copy_connections` the assertion "old network is in form0" does not always hold (however, this is also not necessary).
19. (A) 15.11., 10:38
    Ditto: Computation of `target.a` has an error analogous to error 17.
20. (T) 15.11., 11:11
    In `a_MERGE_connection` upon assignment to `_oe.pe` it must be `<< lxprocN` instead of `<< yprocN`. (Two errors in one identifier!)
21. (D) 16.11., 9:28
    In `CONNECT_Weight`: First use of `base1,base2` is before their initialization (we must either have `_base1,_base2` there or the assignments to `base`X and `basebase`X must be executed before the `if`).
22. (A) 16.11., 9:53
    Ditto: once again a brain-dead loop of the form
    `for (k = _sgl(‘interf_D1.con_ls); k <= 0; k-- ....` (plus another analog one for `interf_D2` plus further two such pairs.
23. (A) 16.11., 10:31
    Ditto: in `‘reorganizeX` we must not have `&new_cons` at the `rsend`, but just `new_cons` instead, because this is already a pointer.
24. (A) 16.11., 11:35
    Ditto: in `‘write meet_rcons and read meet_cons` the normalization is wrong in the assignment to `global_conI`:
    `(norm_p_nodeI-slice1.min) * cons_per_node2 + ‘nd_D2.meI;`
    the correct form would have been
    `norm_p_nodeI * cons_per_node2 + (‘nd_D2.meI - slice2.min);`
25. (A) 16.11., 11:41
    Ditto: in `‘write meet_cons` we must do a `cons++` after setting `meet_cons`, in order to proceed. (For `‘write meet_rcons` and `read meet_cons` and `‘read meet_rcons` this is not necessary, because `cons->_me_D.exists` is being initialized there).
26. (L) 18.11., 13:48
    In `ReductionNodes/WtaNodes` at `‘target addr` it should be
    `(plural void* plural)(x + itarget)` (since `x` is a `_type_` pointer) instead of
    `(plural void* plural)x + (itarget * sizeof (_type_))`, because `sizeof(void)` is 0, so that the addition does not do anything.
27. (A) 19.11., 11:31
    In `NodeArrayInit` at `‘allocate and initialize nodes`: There is an existing node with `meI == arrsizeN` (instead of only up to `arrsizeN-1` as it should be.)
28. (D) 18.11., 11:44
    In `INIT_node` the `ME->interface_D.boss` is not set.
29. (D) 23.11., 10:04
    In `layout__` the `nd->_me_D.boss` is not set for non-existing nodes.
30. (D) 23.11., 10:20
    In `copy_connections` the `con_D.boss` is not set.
31. (D) 23.11., 10:41
    In `CONNECT_` at `‘make con`: `cons->_me_D.boss` is not set; (correspondingly also `rcons->_me_D.boss` at `‘make rcon`).

32. (M) 25.11., 9:56
In `compute_block_layout` the variable `free_enum` must be `int` instead of `_sint` (because it needs to hold the value `-1`). The same is true for variable `layout`.
33. (A) 25.11., 11:06
In `compute_block_layout`: `cum_size -= procN` is wrong if we had waste before. Only the really assigned total size must be subtracted, that is, `com_size -= proc[high].cum_size`.
34. (A) 25.11., 11:11
In `compute_block_size` at `'find high: if (iproc >= low` makes PE 0 always get a `!fits` in the second and later iterations and thus we have `high == -1`.
35. (A) 25.11., 11:40
In `compute_block_layout`: Oh dear, the logic for storing the data about the blocks found in the node descriptions was *completely* wrong. We need intermediate variables for `x0,y0` that are used according to the enumeration of free blocks of size `_S(lbs)` and the nodes of that size (between `low` and `high`).
36. (L) 29.11., 13:38
For an unsigned variable `x` one should not attempt a `for`-loop of the kind `for (x = 10; x >= 0; x--)` since that will never terminate! (Happened in `compute_block_layout`: `'layout one virtualization layer`)
37. (C) 29.11., 13:51
`ss_xsendc` was converted from `dist` to `ldist` only incompletely.
38. (D) 30.11., 16:40
`compute_block_layout` distributes data over all PEs instead of only over segment 0.
39. (A) 30.11., 16:53
`copy_connections` does not test for non-existing nodes or connections and therefore partially works with nonsense data.
40. (L) 03.12., 14:24
In `CONNECT_xx` `'inferf1` is wrong: instead of `(plural _type_*)*(plural char**)(base1+interf_offset1)` it should be `*(plural _type_* plural*)(base1+interf_offset1)`. The first expression should be illegal, but is not flagged as an error by the MPL compiler! (Ditto for `'interf2`). The resulting garbage code also led to garbage results.
41. (D) 03.12., 16:18
In `REPLICATE_Net` the `boss` pointers must not only be changed in `node_group_D` but also in `node_D`! We now do both in the `layout__` procedure.
42. (L) 03.12., 16:36
In `copy_connections` we have the same garbage as in error 40:
`plural char* con = *(plural char**)(old_nd + interface_offset));` should have been
`plural char* con = _sgl(*(plural char* plural*)(old_nd + interface_offset));`
43. (A) 03.12., 16:49
In `compute_block_sizesA` the `(work + (work_per_PE>>1)) / work_per_PE)` results in a division by zero. Just add 1.
44. (L) 07.12., 13:07
In `CONNECT_` at `'reorganize1` we wrongly say
`sp_rsend (cons->_oe.pe, (plural char* plural)new_cons,` instead of
`sp_rsend (cons->_oe.pe, (plural char*)&new_cons,`. A corresponding error is in `'reorganize2`.
45. (D) 08.12., 10:23
In `CONNECT_` at `'write meet_cons` the
`global_conI = ('nd_D1.meI-slice1.min) * cons_per_node2 + norm_p_nodeI;` is wrong, because `cons_per_node2` is the size of slice 1. What we really need is the size of slice 2! A corresponding error is in `'write meet_rcons and read meet_cons` and in `'read meet_rcons`.
46. (D) 08.12., 10:28
In `CONNECT_` at `'write meet_rcons and read meet_cons` we wrongly step `nbpnI` with stepsize `_S(lblocksize1)` (like in `'write meet_cons`) instead with `_S(lblocksize2)`.

47. (A) 08.12., 11:24
In `CONNECT_` at ‘`write meet_cons`, ‘`write meet_rcons and read meet_cons` and ‘`read rcons` the test `if (global_conI < cons_neededN)` makes the procedure use data from non-existing partner nodes. Therefore, the test has to be extended by `&& norm_p_nodeI < cons_per_nodex`. In order to avoid the confusion resulting from `cons_per_node1 == size(slice2)` (where I am interested in the latter) I introduce `slice1size,slice2size`.
48. (M) 08.12., 12:55
When calling `CONNECT_`, the code generation does not make sure that the first group is the local one and the second group is the one with the `_remote_connections`. (Error in code2.fw: `rConnectTo`, `rDisconnectFrom`)
49. (L) 09.12., 09:43
In `PTGremoteFetchCode/PTGremoteSendCode` we use `&ME` as an argument for `rfetch/rsend`, although `ME` is itself a pointer.
50. (A) 09.12., 10:51
In `compute_block_layout` the test for free blocks must be
`(ixproc & _M(lxbs)) == 0 && (iyproc & _M(lybs)) == 0)` (which essentially is *modulo*) instead of `(ixproc & _S(lxbs)) == 0 && (iyproc & _S(lybs)) == 0)`. Immediately after this test we also must check for `free_blocks_enum != -1` in addition to
`free_blocks_enum < relevant_nodesN` or we must initialize `free_blocks_enum` with a large positive value.
51. (L) 09.12., 11:35
Aaaaarghhh!!! In `layout__` I wrote `if (nd->_me_D.exists = _existing)`.
52. (A) 09.12., 12:41
In `compute_work_conI_conN` the computation of `your_xPE,your_yPE` is completely wrong and the second loop condition must be removed since we want a complete prefix sum and not only a reduction. Furthermore, the summation step must not read from `help`, but from `sum` instead.
53. (A) 09.12., 14:46
`reconnect_connections` should have a parameter `descr_offset` and use it in order to check the existence of the connections to be reconnected. Otherwise we often perform very time-consuming blind work: all non-existing connections fetch from PE 0.
54. (X) 13.12., 15:31
Oh dear: in rts.fw (run time system) the functions had `#define rti` set while the rest of `nn3.m` did not. This is where the strange values in the `work` field of some connection descriptors came from. This was probably also the cause of the crashes. At least the program runs without crashing after errors 54 and 55 were removed.
55. (L) 13.12., 15:47
`p_malloc` seems to return 0 when `size == 0`. This confused my `getmem`.
56. (M) 14.12., 12:06
The code generation must not turn a `REPLICATE net INTO 7` into `REPLICATE_Mlp (...., false)` but should output `REPLICATE_Mlp (...., 7 != 1)` instead.
57. (A) 17.12., 10:30
The algorithm of `_INITRANDOM` was nonsense. It generated lots of identical initializations on different PEs. Rewrote it completely.
58. (A) 17.12., 12:56
For expression of the kind `net.field` and `net[i].field` we generated faulty code.
59. (A) 17.12., 14:13
`Reduction_Nodes` did not properly obey `slice`.
60. (A) 17.12., 14:57
In `MergeNet` the refinements of index computation are wrong. Instead of
```
‘y_neighbor_I:    ((ixproc) >> lxN) + (((iyproc+step) >> lyN) << lxN)
```
it should be
```
   ((ixproc) >> lxN) + (((iyproc+step) >> lyN) << (lxprocN-lxN)).
```
61. (A) 17.12., 17:07
Corresponding errors are in `MergeNode` and `MergeCon`. I could have thought of that earlier...

62. (A) 17.12., 18:40
In `layout__` at `‘distribute _existing node to rest of node block` we must conditionalize `nd->_me_D.exists = _shadow;` by an `if (ex = _existing)`, otherwise we generate spurious nodes at strange places.
63. (M) 18.12., 11:50
Aaaarghhhh!! In `test.nn` I had a prototype `Real FUNCTION fabs (Real CONST x) IS EXTERNAL` which does not work as expected, because this function expects `double` arguments! Replaced by `absReal`.
64. (A) 18.12., 13:50
In `layout__` at
`‘fetch _existing old node to each new node and allocate connection arrays` or the corresponding `‘send...` the PE number computation is wrong: `xI` and `yI` must be multiplied by `xblocksize` and `yblocksize`, respectively.
65. (L) 18.12., 14:15
Aarghhhh again! Lutz Prechelt Bugs proudly present: `if (ex = _existing)` in `layout__`:`‘distribute _existing node to rest of node block`. Great. Perhaps I should retire...
66. (A) 18.12., 15:25
`INPUT_x` and `OUTPUT_x` performed index transformation according to `slice` only incompletely.
67. (D) 18.12., 16:08
We *do* need a `a_MERGE__remote_connection`, because otherwise there is no procedure that distributes the descriptor replicates.
68. (P) 18.12., 16:31
In `a_MERGE_Con` the test of `exists` on the outer level successfully suppresses all useful activity as long as the replicates do not yet exist. Analog in `a_MERGE__remote_connection`.
69. (X) 8.1., 16:22
In fileinout.fw (extra module not shown in this document) in procedure `getExamples` at `*firstI = (*firstI + howmany) % exmplN` I had `exN` instead as the last term, which was an uninitialized local variable. It seems this variable happened to have small values always, because otherwise nothing would ever have worked. Probably several strange behaviors can be explained by this error.
70. (A) 12.1., 9:40
In `layout__` upon introduction of `is_replicate0/willbe_replicate0` in `‘fetch _existing old node to each new node and allocate connection arrays` I accidentally put `‘allocate connection arrays and set interfaces boss pointer` outside of the loop `for (i = 0; i < layersN; i++, nd++)`. This made all connections dissappear.
71. (P) 13.1., 15:30
In `ss_xsendc` the `ldist_x` and `ldist_y` were used as if they had the correct value on all PEs. They are valid, however, only on PEs that have `got_it==true` and have to be distributed first.
72. (F) 15.1., 15:20
In semantic analysis at `FieldUse` the `NoKeyMsg` was missing.
73. (P) 15.1., 17:17
In `layout__` at `‘allocate connection arrays and set interfaces boss pointer` the `if (layer == i)` has to be extended by `&& iproc < new_nodesN`, because otherwise we may use uninitialized values.
74. (1) 16.1., 15:02
In `ReplicateNode.tpl` we must have `new_nodesN = reduceMax16u (newI) + 1` (I missed the `+1`). This error later made `layout__` create a node that did not receive a descriptor because no layout information had been computed for it.
75. (M) 10.2., 12:00
When replicating empty node groups `compute_block_sizes` performs a division by zero. Handle this trivial case in `layout__`.
76. (X) 10.2., 13:00

For node procedure calls using slices wrong code was generated (`object.parcode` instead of `object.smallcode`).

77. (A) 11.3., 13:17
In `PTGa[r]ConProcedureDef`: Accessed `ME->_me_D.boss->boss->boss->boss->formA` after termination of the loop `for (... ME++)`.
78. (A) 11.3., 15:07
In `MergeNode.tplr`: Accidentally put an `if (merge && !redistribute)` around the must-be-unconditional call to `a_MERGE_con` when introducing `objs->_cat2(_i,_D).wpc /= repN;`
79. (D) 13.4., 11:00
`delete_connection_postprocessing` must set `_me_D.exists = _nonexisting` for remote connections, but does not.
80. (L) 15.4., 16:08
In `delete_connection_postprocessing` we had
`plural _interface_D* plural remote_interf;` and then
`sp_rsend (oe->pe, (plural char*)&new_conN,`
`(plural char* plural)(remote_interf + offsetof(_interface_D,conN))`. This plus, however is based on `sizeof(_interface_D)`!
81. (M) 16.4., 14:30
In the code generated for `MAXINDEX ME.interface` the call of `update_conI_conN` receives the argument `Object.Type`. This is correct for the dataloc end; the remote end should have `_remote_connection` instead.
82. (L) 18.4., 14:37
In `compute_block_sizesA` we have `plural work1` instead of `plural _work work1`, which creates idiotic distributions as soon as the work exceeds `2**31` (which is less than two minutes).
83. (A) 2.5., 10:02
In `layout__` the argument for the number of nodes parameter of virtualization layer 2 was
`_sgl(net_D->formA) || (_codetype_ == 2 &&`
`_sgl(old_group_D->better2virt) <= 0) ? 0 : newnodesN`. This test of formA is exactly the wrong way round: We want 0 for form0.
84. (1) 2.5., 10:14
When adding nodes in `EXTEND` we compute `meI` for each position of the existing block layout and then determine the new nodes as
`plural _bool new = meI >= group_D->nodesN && meI <= group_D->nodesN+n;` which generates one node too much.
85. (A) 2.5., 17:04
In `reconnect1_connections` at `if (!‘a is in old_cons)` the `!` was missing. This exchanged the cases of intra-group connections and extra-group connections. Chaos was the result for all `REPLICATE_Node` calls.
86. (D) 22.7., 12:38
`delete_connections` does not invalidate `boss->conN`.
87. (1) 11.9., 18:06
In `ReductionNode` and `WtaNode` we wrongly have
`if (slice.max > group_D->nodesN)    slice.max = (plural int)group_D->nodesN - 1;`
where `>=` would be right instead of `>`. This made kogi9 etc. crash.
88. (1) 3.10., 12:34
In `INPUT_` and `OUTPUT_` a similar error is in `‘adjust slice and compute index boundaries;` where both `from` and `to` can only be modified up to the opposite limit. For empty ranges they should go one step farther.
89. (1) 3.10., 12:49
And furthermore the upper limit was `localsizeN` which is 1 too much.

References

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[Knu89] Donald Ervin Knuth. The errors of TEX. *Software — Practice and Experience*, 19(7):607–685, July 1989.

[Pre94] Lutz Prechelt. CuPit — a parallel language for neural algorithms: Language reference and tutorial. Technical Report 4/94, Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany, January 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-4.ps.Z on ftp.ira.uka.de.

[Wil92] Ross N. Williams. *FunnelWeb User's Manual*, version 1.0 for funnelweb 3.0 edition, May 1992.

Index