Vnus language specification

version 2.1 beta 1

C. van Reeuwijk
Delft University of Technology

November 16, 1999

Contents

Chapter 1

Introduction

Vnus is a programming language with a somewhat unusual purpose: it is designed as intermediate language for other programming languages.

It is a programming language because it has well-defined syntax and semantics[1]. It is an intermediate language because it was designed for manipulation by programs, not for easy reading or writing by humans. It *is* possible to write Vnus programs by hand, but this is painful. However, Vnus in its current form it is very convenient as a compilation target and as a representation for automatic manipulation.

Earlier versions of Vnus were designed to be the target language for a compiler for the programming language Booster[3], and therefore copied much of the semantics of that language. The current version is designed to be the target of a Spar compiler. Spar is an experimental programming language that extends Java with language constructs for convenient parallel scientific computations. Although the current version remains compatible with older versions wherever possible, it has not been possible to remain entirely compatible with the earlier language.

Also, a Tm[4, 6] representation of Vnus is now part of the language description, see appendix D. This means that we automatically get a textual representation, a C parser and printer for that textual representation, C data-structure definitions, and C manipulation functions on the C data-structures. Templates for other languages (in particular C++ and Java/Spar) are also available, or are under development.

The grammar of Vnus is formally defined in appendix A. Throughout the text, parts of the Vnus grammar are shown as illustration. For didactic purposes these parts sometimes deviate slightly from the formal definition.

Vnus can be divided into a core language and a number of distinct extensions for different parallel programming models. The core language is more-or-less equivalent to the sequential subset of the language. It is described in chapters 3 and 4. The extensions for parallel programs are described in chapter 5.

[1]In fact, for an older version of the language there is a denotational semantics description[1].

Chapter 2

Grammar notation

2.1 The purpose of the grammar rules

Throughout the document, grammar rules are used to show the grammar of the various Vnus language constructs. They are provided to *describe* the grammar of the language, not to define it.

The syntax of Vnus is *defined* in appendix A.

2.2 The notation

The grammar notation style that is used in this report is very similar to the style used in The Java Language Specification[2]. The grammar consists of a series of *productions*, where each production specifies the derivation of a *nonterminal* symbol from terminals and other nonterminals. Nonterminal symbols are shown in an *italic* font, terminal symbols are shown in a `fixed width` font.

The definition of a nonterminal is introduced by the name of the nonterminal, followed by a colon. One or more alternative production rules then follow on succeeding lines. For example, the production:

> *statement*:
> `if` *expression block block*

states that the nonterminal *statement* produces the terminal `if`, followed by the nonterminal *expression*, and twice the nonterminal *block*.

To clarify the meaning of the language construct, it is allowed to prefix a nonterminal with a name, followed by a dash. This prefix is ignored in the interpretation of the grammar rules. For example, the following production is identical to the previous one:

> *statement*:
> `if` *condition-expression true-block false-block*

A production may have more than one alternative; each alternative is listed on a separate line, or in a separate production. For example, an *expression* has several alternative productions:

> *expression*:
> *identifier*
> `(` *expression* `,` *binop* `,` *expression* `)`

> *expression*:
> `(` *unop* `,` *expression* `)`

The subscripted suffix '*opt*', which may appear after a terminal or nonterminal, indicates an *optional symbol*. This suffix is a shorthand for two grammar rules, one that omits the optional element, and one that includes it. For example:

statement:
`return` $expression_{opt}$

is a shorthand for:

statement:
`return`
`return` *expression*

If a grammar rule contains more than one optional symbol, all combinations of presence and absence of the symbols are allowed.

The subscripted suffix '*list*', which may appear after a terminal or nonterminal, indicates a *list*. It replaces separate productions to specify a comma-separated list of instances of the subscripted symbol.

For example, the production:

statement:
`record [` $expression_{list}$ `]`

is a shorthand for:

statement:
`record [` $expressionList_{opt}$ `]`

expressionList:
expression
expressionList `,` *expression*

When all the alternative productions of a nonterminal consist of a single terminal symbol, a more compact notation is used, where all alternatives are listed on a single line. For example:

unary: one of
`not + ~`

Chapter 3

Basic language constructs

This chapter describes the basic language constructs of Vnus. These constructs form the sequential and scalar subset of Vnus. Shapes, records, data-parallel extensions, and explicitly parallel extensions are discussed in separate chapters.

3.1 Program structure

This is the smallest possible Vnus program:

```
program
declarations []
statements []
```

In general, Vnus programs must have the following grammar:

program:
    `program` *globalPragmas*$_{opt}$ *declarations main-block*

globalPragmas:
    `pragma` *pragmas*

A Vnus program consists of two sections that *must* be present (even if empty), and that *must* be provided in the shown order. The *declarations* section contains function, procedure, and variable declarations. It lists the type, scope and other relevant information for all symbols (variables, functions, and procedures) that occur in the program. Scope names, statement labels, and the field names of each record all have a separate name space, and are not declared.

The *main-block* section contains the list of statements that must be executed; it can *not* be used as the scope of a local variable, since it does not have a scope name associated with it, therefore only global declarations are visible in the statement list.

A program may have global pragmas. These are specified immediately after the `program` keyword, as shown in the grammar. See section 3.13 for more information about pragmas.

3.2 Comment

Vnus supports two forms of comment: *line comments*, that start with '`//`', and continue to the end of the line; and *multi-line comments*, that start with '`/*`' and end with '`*/`'. Nested comment is not supported. In particular, the sequence `/* /* */ */` is scanned as a comment up to, and including the *first* `*/` sequence.

The following program demonstrates the use of comment:

```
// This is a line comment
program
```

```
declarations [
/*
 * And this is a multi-line comment.
 */
]
statements []
```

3.3 Base types and literals

Vnus supports the following basic types[1]:

type	bytes	range
`byte`	1	-2^7 to 2^7-1
`short`	2	-2^{15} to $2^{15}-1$
`int`	4	-2^{31} to $2^{31}-1$
`long`	8	-2^{63} to $2^{63}-1$
`char`	2	-2^{15} to $2^{15}-1$
`string`	variable	n.a.
`float`	4	see text
`double`	8	see text

The types `byte`, `short`, `int`, and `long` are collectively called the *integral types*. The types `float` and `double` are collectively called the *floating point types*. The floating point types and the integral types are collectively called the *numeric types*.

The floating point types are `float` and `double`, representing single-precision 32-bit and double-precision 64-bit format IEEE 754. Operations on these values also follow the IEEE 754 standard[2].

Next to the normal floating-point numbers, the IEEE 754 standard also includes positive and negative zeros, positive and negative infinities, and a special value named *Not a Number* (*NaN*). In Vnus the infinities are represented by `inf`, and `-inf`; `+inf` is also accepted, and is equivalent to `inf`. Not-a-Number is represented by `nan`, it may be preceded by a sign, which is ignored.

The finite nonzero values of type `float` are of the form $s \cdot m \cdot 2^e$, where s is $+1$ or -1, m is a positive integer less than 2^{24}, and e is an integer between -149 and 104, inclusive. Values of that form such that m is positive but less than 2^{23} and e is equal to -149 are said to be *denormalized*.

The finite nonzero values of type `double` are of the form $s \cdot m \cdot 2^e$, where s is $+1$ or -1, m is a positive integer less than 2^{53}, and e is an integer between -1075 and 970, inclusive. Values of that form such that m is positive but less than 2^{52} and e is equal to -1075 are said to be *denormalized*.

Except for NaN, floating-point values are *ordered*; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, negative zero, positive zero, positive nonzero values, and positive infinity.

Positive zero and negative zero compare equal; thus the result of the expression `(0.0,=,-0.0)` (equality comparison between 0.0 and -0.0 in Vnus syntax) is `true` and the result of `(0.0,>,-0.0)` is `false`. But other operations can distinguish positive and negative zero; for example, `(1.0,/,0.0)` has the value `inf`, while the value of `(1.0,/,-0.0)` is `-inf`.

NaN is *unordered*, so the numerical comparison operators `<`, `<=`, `>`, and `>=` return `false` if either or both operands are `nan`. The equality operator `=` returns `false` if either operand is `nan`, and the inequality operator `<>` returns `true` if either operand is `nan`.

Any value of a floating-point type may be cast to or from any numeric type, and to and from type `char`, see section 3.4.6.

3.4 Expressions and locations

Vnus expressions are like the expressions in most programming languages; they represent values that are determined at compile-time or run-time through evaluation. Every expression is strongly

[1] In a future version of Spar the names of the basic types will be changed to words such as `uint16` and `int32`, since these names are more appropriate for an intermediate language such as Vnus.

[2] Much of the following section is derived from [2].

typed: it is always possible to determine the type of any expression at compile-time[3]. Expressions without function calls and assertions are side-effect free. The order of evaluation of expressions is unspecified. This is only relevant for expressions with side-effects.

The following program demonstrates the use of expressions; it prints 36 to standard output[4]:

```
program
declarations [
    externalvariable vnusout int
]
statements [
    println [vnusout,((3,+,3),*,cast int 6.2)]
]
```

This demonstrates the Vnus addition, multiplication and cast operators.

Vnus locations are similar to expressions, but they represent storage locations. Locations are strongly typed; it is always possible to determine the type of any location at compile-time.

3.4.1 Literals

The simplest form of expression is the *literal*.

All Vnus literals must be typed, and therefore there is a separate literal for each base type.

expression:
  intLiteral
  floatLiteral
  charLiteral
  booleanLiteral
  stringLiteral

For example, the following program prints literals of all base types:

```
program
declarations [
    externalvariable vnusout int
]
statements [
    println [vnusout,"Integral: byte:",1b," short:",2s," int:",4," long:",8l],
    println [vnusout,"Floating point: float:",4.0f," double:",8.0d],
    println [vnusout,"Character: ",'?'],
    println [vnusout,"String: ","hello world"],
    println [vnusout,"All boolean literals: ",true,":",false]
]
```

This program produces the following output:

```
Integral: byte:1 short:2 int:4 long:8
Floating point: float:4.000000 double:8.000000
Character: ?
String: hello world
All boolean literals: TRUE:FALSE
```

3.4.2 The complex constructor

Complex numbers don't have a literal, but are always constructed from `double` expressions:

expression:
  `complex` *real-expression imaginary-expression*

Both expressions must be of type `double`.

[3]There is one exception: the `null` literal has the more general type 'any pointer'.
[4]This program uses the statement `println`, which prints data to an output stream. See section 3.12 for details.

3.4.3 Unary operators

Unary operators have the following syntax:

expression:
  `(` *unary* `,` *expression* `)`

unary: one of
  `not + ~`

The unary operators have the following meaning and application area:

symbol	description	types
`not`	bitwise inversion, boolean negation	integral, boolean
`+`	unary plus	numerical
`~`	negation	numerical

The symbol '`~`' was chosen as numerical negation operator to distinguish between the `-` sign in negative constants and this operand.

3.4.4 Binary operators

expression:
  `(` *expression* `,` *binary* `,` *expression* `)`

binary: one of
  `* + - / < << <= <> = > >= >> >>> and mod or xor`

The binary operators have the following meaning and application area:

symbol	description	types
`*`	multiplication	numerical
`+`	addition	numerical, string
`-`	subtraction	numerical
`/`	division	numerical
`<`	less	numerical, string
`<<`	shift left	integral
`<=`	less or equal	numerical, string
`<>`	not equal	numerical, string, pointer
`=`	equal	numerical, string, pointer
`>`	greater	numerical, string
`>=`	greater or equal	numerical, string
`>>`	shift right	integral
`>>>`	unsigned shift right	integral
`and`	bitwise or boolean 'and'	integral, boolean
`or`	bitwise or boolean 'or'	integral, boolean
`xor`	bitwise or boolean 'exclusive or'	integral, boolean

Apart from the shift operators, all binary operator require two operands of *exactly* the same type. For example, you cannot add a `short` and an `int`. You must use a cast operator (see section 3.4.6) to convert one of the two operands to the type of the other.

The shift operators require an integral type on the left-hand side, and an `int` type on the right-hand side.

The `and` and `or` operators on boolean expressions use 'shortcircuit' evaluation. That is, first the lefthand side of the operator is evaluated. For the `and` operator, the righthand side is only evaluated if the lefthand side evaluates to `true`. For the `or` operator, the righthand side is only evaluated if the the lefthand side evaluates to `false`.

In all other cases all operands are evaluated, and the order of evaluation is unspecified.

3.4.5 The conditional operator

The Vnus conditional expression has the following syntax:

expression:
`if` *cond-expression true-expression false-expression*

The first expression is the condition expression; it must be of type `boolean`. The other two expressions are the result for a `true` and `false` condition, respectively; both must be of the same type.

For example, the following program prints `yes` :

```
program
declarations [
    externalvariable vnusout int
]
statements [
    println [vnusout, if (3,<,4) "yes" "no"]
]
```

3.4.6 Casts

Vnus does *not* do any implicit type conversion *at all*: every type conversion must be expressed as a cast expression.

expression:
`cast` *type expression*

For example, the following program prints 42:

```
program
declarations [
    externalvariable vnusout int
]
statements [
    println [vnusout, (7,*,cast int 6s)]
]
```

All conversions between numerical types are allowed. In some cases the conversion leads to loss of precision. Conversion between longs and doubles, and between ints and floats is done based on their value, not by interpreting the bits in a different way. Rounding behavior is system-dependent (i.e. it does what your C implementation does).

A `char` can be cast to a numerical type, resulting in the Unicode code of the character; numerical types can be cast to a `char`, resulting in the character corresponding to the given Unicode code.

Booleans, complex numbers and strings can only be cast to the same type.

Casts of constructed types will be discussed in chapter 4.

3.5 Variables, symbols, and assignment

As in many other languages, all Vnus variables must be declared, but in Vnus this is done in a somewhat unusual way: all declarations are grouped together in the `declarations` section. Every variable, function and procedure in a Vnus program must have exactly one entry in the declaration section. As a consequence, all variable names—including local variables, formal parameters, and cardinality variables—must be unique, and cannot be used in two different scopes, even if the scopes do not overlap.

What information a declaration contains depends on the category of the variable. For example, for global variables the syntax of the entry is:

declaration:
  `globalvariable` *identifier type init-expression*$_{opt}$ *modifiers*$_{opt}$ *pragmas*$_{opt}$

modifiers:
  modifier
  modifiers modifier

modifier: one of
  `const local unchecked volatile`

The meaning of *pragmas* is explained in section 3.13. The modifier `const` indicates that the value of the variable will not be changed. The declaration of a `const` variable must have an initializer. The modifier `unchecked` indicates that accesses to this variable need not be checked for range errors. This is only useful for shape types, see chapter 4. This modifier is typically used when the front-end already generates `Vnus` code to do the bounds checking. The modifier `volatile` is discussed in section 5.3.3. The `local` modifier indicates that a variable is local to every processor, and therefore does not have to be kept synchronized with instances on other processors.

If there is an *init-expression*, the variable is initialized with that value. The expression must be constant.

Variables can be used in expressions and locations in the obvious way:

expression:
  identifier

location:
  identifier

The following program demonstrates the use of an initialized global variable 'v'; it will print 1764:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable v int 42
]
statements [
    println [vnusout,(v,*,v)]
]
```

Provided that the variable is not declared `const`, it can be assigned to with the assignment statement, which has the following syntax:

statement:
  `assign` *dest-location src-expression*

Assignment is only allowed when the type of the source and the destination are equivalent. Base types are only equivalent to themselves, but for constructed types the equivalence relation is more complicated.

The following program demonstrates the use of the assignment statement; it will print 81:

```
program

declarations [
    externalvariable vnusout int,
    globalvariable v int
]
statements [
    assign v 7,
    assign v (v,+,2),
    println [vnusout,(v,*,v)]
]
```

3.6 Blocks, scopes, and local variables

Like many other programming languages, Vnus supports variables that are only active within a restricted range of the program. Such an activation range is called the *scope* of the variable. In Vnus the scope of a variable is either the entire program (for global and external variables, and for functions and procedures), or a group of statements called a *block* (for all other classes of variables). The syntax of a statement block is:

block:
  `statements` $scopename_{opt}$ $pragmas_{opt}$ `[` $labeledStatement_{list}$ `]`

labeledStatement:
  $label_{opt}$ $pragmas_{opt}$ *statement*

Blocks can be used at any place a single statement is allowed:

statement:
  block

Every block can have a scopename. It is used in the symbol table to specify the scope of local variables and formal parameters. For local variables the syntax of a symbol table entry is:

declaration:
  `localvariable` *identifier scopename type* $init\text{-}expression_{opt}$ $modifiers_{opt}$ $pragmas_{opt}$

The following program demonstrates the use of local variables:

```
program
declarations [
    externalvariable vnusout int,
    localvariable v outerscope int,
    localvariable w innerscope int
]
statements [
    statements outerscope [
        statements innerscope [
            assign w 7,
            assign v (w,+,1),
            println [vnusout,"v: ",v," w: ",w]
        ],
        assign v (v,+,1),
        // ‘w’ is not visible any more
        println [vnusout,"v: ",v]
    ]
]
```

it will produce the following output:

```
v: 8 w: 7
v: 9
```

3.7 Control statements

3.7.1 The `for` statement

The `for` statement is like the `for` statement of many programming languages: it will iterate over the body of the iteration statement for the given number of times. An iteration variable (called a *cardinality variable* in Vnus), is provided to the iteration body.

Compared to iteration statements in many other programming languages, Vnus iterations are generalized, because they allows iteration over a list of variables.

Iteration statements have the following syntax:

statement:
`for` *cardinalities body-block*

cardinalities:
`[` $cardinality_{list}$ `]`

cardinality:
variable-identifier `=` *low-expression* `:` *up-expression* $secondaries_{opt}$
variable-identifier `=` *low-expression* `:` *up-expression* `:` *stride-expression* $secondaries_{opt}$

secondaries:
*'(' * $secondary_{list}$ *')'*

secondary:
variable-identifier `=` *low-expression* `:` *stride-expression*

declaration:
`cardinalityvariable` *identifier* $modifiers_{opt}$ $pragmas_{opt}$

A cardinality with lower bound l and upper bound u specifies that the cardinality variable iterates over $l, l+1, \ldots u-1$. For example, the program

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i
]
statements [
    for [i=2:5] statements [
        print [vnusout,i,' ']
    ],
    println [vnusout]
]
```

The program will produce the following output:

```
2 3 4
```

The bounds of the cardinality must be `int` expressions. Every cardinality variable must be declared as such. It cannot be used at any other place as a cardinality or secondary. It is not allowed to assign to cardinality variables.

If a stride s is specified, the iteration iterates over $l, l+s, \ldots u-1$. For example, the program

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i
]
statements [
    for [i=2:12:3] statements [
        print [vnusout,i,' ']
    ],
    println [vnusout]
]
```

The program will produce the following output:

```
2 5 8 11
```

The stride of a cardinality variable must be an `int` expression, and must not be negative or zero.

A *secondary* is a variable that *follows* the iterations of the cardinality variable. For every iteration number i, a secondary variable with lower bound l_s and stride s_s has the value $l_s + i \cdot s_s$ where the cardinality variable has value $l + i \cdot s$. For example, the program

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
    for [i=2:6 (j=5:-1)] statements [
        print [vnusout,i,'~',j,' ']
    ],
    println [vnusout]
]
```

The program will produce the following output:

```
2~5 3~4 4~3 5~2
```

The lower bound and stride of a secondary variable must be an `int` expression; there are no restrictions on their values. Every secondary variable must be declared as a cardinality variable; it cannot be used at any other place as a cardinality or secondary variable. It is not allowed to assign to secondary variables.

A cardinality can have an arbitrary number of secondaries. For example, the following program shows a cardinality with two secondaries:

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i,
    cardinalityvariable j,
    cardinalityvariable k
]
statements [
    for [i=0:12 (j=0:2,k=12:-1)] statements [
        println [vnusout,"i=",i," j=",j," k=",k]
    ]
]
```

The program will produce the following output:

```
i=0 j=0 k=12
i=1 j=2 k=11
i=2 j=4 k=10
i=3 j=6 k=9
i=4 j=8 k=8
i=5 j=10 k=7
i=6 j=12 k=6
i=7 j=14 k=5
i=8 j=16 k=4
i=9 j=18 k=3
i=10 j=20 k=2
i=11 j=22 k=1
```

A *cardinality list* specifies an iteration volume for the vector of iteration variables. The volume need not be rectangular, since bound expressions in a cardinality list may contain cardinality variables that are visible in its scope. For the sake of regularity the empty cardinality list is accepted. In this case, all Vnus iterator statements degenerate to a single, sequential, execution of the body `block`.

The scope of every cardinality and secondary variable is all cardinality expressions to the right of it, and the body of the iteration. Therefore, a cardinality variable can be used in the upper-bound expression of the cardinality expressions to the right of it. Cardinality variables can be used in only one cardinality. It is not allowed to assign to cardinality variables.

All iteration statements enumerate the iteration volume specified by the cardinality list. In the `for` statement, the iteration volume is scanned in a specified order: the enumeration order is equivalent to the enumeration order of a series of nested `for` statements, where the rightmost cardinality is put in the innermost loop. Consider, for example, the following program:

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
    for [i=0:4,j=0:3] statements [
        print [vnusout, i,",",j," | "]
    ],
    println [vnusout]
]
```

This program will produce the following output:

```
0,0 | 0,1 | 0,2 | 1,0 | 1,1 | 1,2 | 2,0 | 2,1 | 2,2 | 3,0 | 3,1 | 3,2 |
```

By definition, the execution order of the `for` statement in this program is equivalent to that of the `for` statements in the following program:

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
    for [i=0:4] statements [
        for [j=0:3] statements [
            print [vnusout,i,",",j," "]
        ]
    ],
    println [vnusout]
]
```

See Section 5.3 for a description of the `foreach` iterator, and Section 5.2.3 for a description of the `forall` iterator.

The following program demonstrates the use of a cardinality variable in a subsequent cardinality range expression.

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
```

```
    for [i=0:4,j=0:(i,+,1)] statements [
        print [vnusout, i,",",j," | "]
    ],
    println [vnusout]
]
```

The program will produce the following output:

```
0,0 | 1,0 | 1,1 | 2,0 | 2,1 | 2,2 | 3,0 | 3,1 | 3,2 | 3,3 |
```

3.7.2 The `if`, `while` and `dowhile` statements

The `if`, `while` and `dowhile` statements have the following syntax:

statement:
`while` *expression block*

statement:
`dowhile` *expression block*

statement:
`if` *cond-expression true-block false-block*

They have the expected semantics: the *block* of the `while` statement is executed while the condition expression evaluates to `true`; the *block* of the `dowhile` is executed at least once until the condition expression evaluates to `false`; in the `if` statement the condition expression is evaluated, and if the result is `true`, the first *block* is executed, else the second *block* is executed. The condition expression of the `while`, `dowhile` and `if` statement must be of type `boolean`.

The following program demonstrates the use of the `if`, `while`, and `dowhile` statements:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable v int
]
statements [
    if (3,<,4) statements [
        println [vnusout,"yes"]
    ]
    statements [
        println [vnusout,"no"]
    ],
    assign v 3,
    while (v,>,0) statements [
        println [vnusout,"v=",v],
        assign v (v,-,1)
    ],
    dowhile (v,>,0) statements [
        println [vnusout,"v=",v],
        assign v (v,-,1)
    ]
]
```

The program will produce the following output:

```
yes
v=3
v=2
v=1
v=0
```

3.7.3 The `switch` statement

The `switch` statement has the following syntax:

statement:
`switch` *expression* `[` $switchCase_{list}$ `]`

switchCase:
`(` *intLiteral* `,` *block* `)`
`(` `default` `,` *block* `)`

The expression to switch on must be of type `int`.

This switch statement is very similar to the Java and C `switch` statements: the switch expression is evaluated, and control is transferred to the statement block that is labeled with that value. If no block is labeled with the found value, control is transferred to the default block.

Every block 'falls through' to the next block in the case list. The last block falls through to the default block.

The following program demonstrates the use of the `switch` statement:

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable v
]
statements [
    for [v=0:4] statements [
println [vnusout,"v=",v],
        switch v [
            (0, statements [
                println [vnusout," zero"]
            ]),
            (1, statements [
                println [vnusout," one"]
            ]),
            (2, statements [
                println [vnusout," two"]
            ]),
            (default,statements [
                println [vnusout," default"]
            ])
        ]
    ]
]
```

The program will produce the following output:

```
v=0
 zero
 one
 two
 default
v=1
 one
 two
 default
v=2
 two
```

```
 default
v=3
 default
```

3.8 Functions and procedures

Functions and procedures are declared in the `declaration` section of a Vnus program. The syntax is as follows:

declaration:
  `procedure` *identifier formalParameters modifiers*$_{opt}$ *pragmas*$_{opt}$ *block*
  `function` *identifier formalParameters return-type modifiers*$_{opt}$ *pragmas*$_{opt}$ *block*

formalParameters:
  `[` *formalParameter*$_{list}$ `]`

formalParameter:
  identifier

The formal parameters of functions and procedures must also be declared, using the following syntax.

declaration:
  `formalvariable` *identifier type scopename modifiers*$_{opt}$ *pragmas*$_{opt}$

Functions can only be invoked in expressions, using the following syntax:

expression:
  `functioncall` *routineExpression actualParameters*

actualParameters:
  `[` *actualParameter*$_{list}$ `]`

actualParameter:
  expression

routineExpression:
  identifier
  `*` *expression*

Procedures can only be invoked in the procedure call statement, using the following syntax:

statement:
  `procedurecall` *routineExpression actualParameters*

Parameters are always passed by value. Passing by reference must be emulated with pointer parameters, see section 4.3. When a function or procedure is activated, its formal parameters are instantiated by assigning the actual parameters to them. All restrictions and semantics that are associated with assignment statements also apply to this implicit assignment.

The following program demonstrates the use of functions and procedures; the meaning of the `return` statement is explained in section 3.10.

```
program
declarations [
    externalvariable vnusout int,
    procedure min4 [a, b, c, d] statements min4scope [
        assign mn functioncall min [
            functioncall min [a, b],
            functioncall min [c, d]
```

```
        ],
        println [vnusout,"Minimum: ",mn]
    ],
    localvariable mn min4scope int,
    formalvariable a min4scope int,
    formalvariable b min4scope int,
    formalvariable c min4scope int,
    formalvariable d min4scope int,
    function min [mina, minb] int statements minscope [
        if (mina,<,minb) statements [
            return mina
        ]
        statements [
            return minb
        ]
    ],
    formalvariable mina minscope int,
    formalvariable minb minscope int
]
statements [
    procedurecall min4 [1, 2, 3, 4],
    procedurecall min4 [1, 44, -3, 8]
]
```

The program will produce the following output:

```
Minimum: 1
Minimum: -3
```

3.9 External declarations

In principle, a Vnus program must contain definitions for all variables, functions, and procedures that are used in the program; Vnus does not support separate compilation. However, there are a few circumstances where a Vnus program will refer to external definitions. In particular, a Vnus program may refer to a library of standard routines.

To ensure proper type checking of external functions, procedures, and variables, Vnus requires an entry in the symbol table for each of them.

declaration:
  `externalvariable` *identifier type modifiers*$_{opt}$ *pragmas*$_{opt}$
  `externalfunction` *identifier formalParameters return-type modifiers*$_{opt}$ *pragmas*$_{opt}$
  `externalprocedure` *identifier formalParameters modifiers*$_{opt}$ *pragmas*$_{opt}$

3.10 Jumps

The `return` and `goto` statements change the flow of control in an unstructured manner.

The `return` statement is provided to allow abrupt termination of the execution of a function or procedure. It is also used to specify the value that is returned by a function. It has the following syntax:

statement:
  `return` *expression*$_{opt}$

A return statement with no *expression* can only occur in a procedure. It transfers control to the invoker of the current procedure. A `return` statement with an *expression* can only occur in a function. The type of the *expression* must be the same as the return type of the function. The statement evaluates the expression, assigns the result to the return value of the expression, and transfers control to the invoker of the function.

The `goto` statement transfers control to the statement with the given label. It has the following syntax:

> *goto*:
> `goto` *labelName*

It is not allowed to transfer control to a scope that is nested in the current scope, nor is it allowed to transfer control outside the current function or procedure. Moreover, it is not allowed to transfer control to another statement in a `each`, `foreach`, `fork`, or `forkall` statement.

All statement labels within a function or procedure must be unique. All global statement labels must also be unique.

The following program demonstrates the use of the `return` and `goto` statements.

```
program
declarations [
    externalvariable vnusout int,
    globalvariable v int,
    function min [mina, minb] int statements minscope [
        if (mina,<,minb) statements [
            return mina
        ]
        statements [
            return minb
        ]
    ],
    formalvariable mina minscope int,
    formalvariable minb minscope int
]
statements [
    println [vnusout,functioncall min [3,4]],
    println [vnusout,functioncall min [3,-4]],
    assign v 0,
    switch (v,+,1) [
        (0, statements [
            println [vnusout,"zero"],
            goto endsw
        ]),
        (1, statements [
            println [vnusout,"one"],
            goto endsw
        ]),
        (2, statements [
            println [vnusout,"two"],
            goto endsw
        ]),
        ( default, statements [
            println [vnusout,"default"]
        ])
    ],
endsw:
    println [vnusout,"done"]
]
```

The program will produce the following output:

```
3
-4
one
done
```

3.11 Exceptions

The exception handling statements have the following syntax:

> *statement*:
> `throw` *expression*
> `rethrow`
> `catch` *formalParameter try-block handler-block*

The `throw` statement throws an exception with the given expression as value. Only pointers to records can be thrown. Control is passed to the handler block of the innermost activate `catch` statement. If there is no active catch statement, the default exception handler is activated, which terminates execution of the entire program.

The `catch` statement executes the code in the *try* block. During execution of the *try* block the statement is said to be a *active*. If during execution of the *try* block an exception is thrown, and if the exception is not caught by a nested active `catch` statement, the exception is caught by this `catch` statement. When an exception is caught, the value of the exception is assigned to the formal parameter of the `catch` statement, and the *handler* block is executed.

If the try block terminates without throwing an exception, the handler block is not executed.

The `rethrow` statement is only allowed in the *handler* block of a `catch` statement. It throws the value that was assigned to the formal parameter of the `catch` statement.

Since a `catch` statement always catches *all* exceptions, the handler usually will contain some form of decoding to distinguish between the possible types of exception. In Vnus there is no special support for this, however.

The following program demonstrates the use of exception statements:

```
program
declarations [
    formalvariable v catchblock pointer record [],
    record r1 [
        id:int
    ],
    globalvariable x pointer record [id:int]
]
statements [
    catch v statements catchblock [
        println [1,1],
        assign x new record[id:int],
        assign field *x id 42,
        throw x
    ]
    statements [
        println [1,2]
    ],
    println [1,3]
]
```

The program will produce the following output:

```
1
2
3
```

3.12 Support statements

Vnus provides a few statements that are difficult to classify, but make life a little easier.

The `empty` statement does nothing. It is useful as target of a `goto` (since there may be no other statement to jump to), and to translate empty statements in front-ends to. The syntax is simply:

> *statement*:
> `empty`

The `print` and `println` statements are mainly intended for debugging, testing, and demonstration purposes. Given a list of parameters, a string is printed to standard output for each of the parameters. For base types the obvious output is generated (e.g. strings are directly printed, numbers are converted to a number string, etc.). For constructed types a short summary of the contents is printed. The `println` statement also prints a newline after the list of parameters has been printed.

> *statement*:
> `print` *actualParameters*

> *statement*:
> `println` *actualParameters*

The first parameter of the `print` and `println` statements must be of type `int`, and denotes the output stream to which the output is written. Two output streams are defined in the external `int` variables `vnusout` and `vnuserr`; they are attached to the standard output and error stream respectively. In parallel Vnus, the second parameter of the `print` and `println` statements must be of type `int`, and denotes the processor the statement is executed on.

The following program demonstrates the `empty`, `print`, and `println` statements:

```
program
declarations [
    externalvariable vnusout int,
    cardinalityvariable i1,
    cardinalityvariable i2
]
statements [
    for [i1=0:5] statements [
        print [vnusout,i1,":"]
    ],
    empty,
    println [vnusout],
    empty,
    for [i2=0:5] statements [
        println [vnusout,i2,":"]
    ],
    empty,
    println [vnusout,"done"]
]
```

The program will produce the following output:

```
0:1:2:3:4:
0:
1:
2:
3:
4:
done
```

3.13 Pragmas

The engines that work on Vnus often require additional information about the language constructs. This information may have been computed by analysis tools, and may be too costly to throw away; it may spring from specific knowledge of a front-end, or perhaps it is an annotation from the user. To represent this information, Vnus supports pragmas in many of its language constructs.

pragmas:
  `pragma [` $pragma_{list}$ `]`

pragma:
  identifier
  identifier `=` *PragmaExpression*

PragmaExpression:
  LiteralPragmaExpression
  NamePragmaExpression
  ListPragmaExpression

LiteralPragmaExpression:
  intLiteral
  floatLiteral
  doubleLiteral
  stringLiteral
  booleanLiteral

NamePragmaExpression:
  identifier
  `@` *identifier*

ListPragmaExpression:
  `(` *PragmaExpressions* `)`

PragmaExpressions:
  empty
  PragmaExpressions PragmaExpression

Pragmas come in two forms: in the first form the mere presence of the pragma with the given identifier is sufficient to convey the information; such a pragma is called a `flag` pragma. In the second form, the pragma requires a value; such a pragma is called a `value` pragma.

In appendix C the pragmas are listed that are recognized by the Vnus to C translator. Other phases of the compilation process may use other pragmas, appendix B lists the pragmas that are of interest in more than one engine.

In previous sections it was already shown that declarations and statements can be annotated with pragmas, and that there is an optional section for global pragmas. It is also possible to annotate a location or expression with pragmas, using the following syntax:

location:
  `expressionpragma` *pragmas location*

expression:
  `expressionpragma` *pragmas expression*

The following program demonstrates the use of pragmas:

```
program
pragma [boundscheck=true,listingfile="pragma.lst"]
declarations [
    externalvariable vnusout int,
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
    pragma [independent,iters=12] for [i=0:4,j=0:3] statements [
        println [vnusout,expressionpragma [range = (0 4)] i," ",j]
    ]
]
```

One pragma statement attaches the `flag` pragma `independent` to the `for` statement. The other pragma statement attaches the `value` pragma statement `range` to the variable expression `i`. The range pragma contains a record with two `int` values.

3.14 Where expressions and locations

expression:
`where` *scopename expression*

location:
`where` *scopename location*

A `where` expression introduces a new scope, wherein local variables can be declared. The given *expression* or *location* is then evaluated within that scope.

This construct is mainly used to avoid double evaluation of an expression, either because the expression has side-effects (through function invocation), or because evaluation is expensive.

The following program demonstrates a typical situation where a `where` expression might be used.

```
program

declarations [
    externalvariable vnusout int,
    globalvariable v int,
    localvariable s wherescope pointer string cast pointer string null
]
statements [
    assign v 7,
    assign v (v,+,2),
    println [
        vnusout,
        where wherescope if (s,=,cast pointer string null) "null" *s
    ]
]
```

This program produces the following output:

```
null
```

Chapter 4

Constructed types

In this chapter all *constructed* types are discussed. That is, all types that are constructed from other types. These types are: shapes, records and pointers.

4.1 Shapes

A shape is group of elements, all of the same type, that can be accessed through an *index* expression. The elements are ordered into a rectangular block with an non-negative number of dimensions, and with a non-negative size in each dimension.

The number of dimensions of the shape is called its *rank*; a shape of rank n is also called a n-dimensional shape. Access into such a shape requires an index expressions with n elements. For example, a shape of rank 2 is said to be two-dimensional, and requires index expressions with two elements.

The syntax of a shape type is as follows:

type:
  `shape` *sizes* $distributions_{opt}$ *type*

sizes:
  `[` $size_{list}$ `]`

size:
  expression
  `dontcare`

The syntax and semantics of *distributions* will be explained in chapter 5.2.

A shape type may contain size expressions, but these only specify the *initial* size of a shape; throughout its lifetime a shape may grow and shrink. The rank of a shape, and the type of its elements, cannot be changed. If one of the size expressions equals `dontcare`, a shape has undefined size. Such a shape can only be used after it has been given a definite size.

To refer to elements of a shape in expressions and locations, the following syntax is used:

location:
  selection

expression:
  selection

selection:
  expression `[` $selector_{list}$ `]`

selector:
  expression

The number of selectors must be equal to the rank of the shape. A selector must be of type `int`.

The following program demonstrates the use of shapes:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable a shape [3,4] int,
    cardinalityvariable i,
    cardinalityvariable j,
    cardinalityvariable k,
    cardinalityvariable l,
    globalvariable sum int 0
]
statements [
    for [i=0:3,j=0:4] statements [
        assign (a,[i,j]) (i,+,j)
    ],
    for [k=0:4,l=0:3] statements [
        assign sum (sum,+,(a,[l,k]))
    ],
    println [vnusout,sum]
]
```

It will produce the following output:

```
30
```

The size of a shape in a given dimension can be retrieved with the operator `getsize`, which has the following syntax:

> *expression*:
> `getsize` *shape-expression dimension-expression*

The first expression must be a `shape`, and the second expression must be of type `int`. The expression returns an `int` that contains the size of the given shape in the given dimension.

The size of a shape can be changed with the `setsize` statement, which has the following syntax:

> *statement*:
> `setsize` *location* `[` $expression_{list}$ `]`

The *location* must refer to a `shape`.

The length of a shape (the product of the sizes in all dimensions) can be retrieved with the operator `getlength`, which has the following syntax:

> *expression*:
> `getlength` *shape-expression*

The expression must be a `shape`, the expression returns an `int` that contains the length of the given shape.

Shape types with the same rank and with equivalent element types are equivalent. Shapes can be assigned if the types are equivalent; the destination shape assumes the size of the source, and the elements of the source shape are assigned to the corresponding elements in the destination shape.

An expression of a shape type can be constructed using the `shape` constructor expression, which has the following syntax:

> *expression*:
> `shape` *sizes element-type* `[` $expression_{list}$ `]`

The sizes define the size of the shape in each dimension, the type is the type of the elements (*not* the type of the entire array). The list of expressions should be as large as the product of all sizes.

Currently, only one-dimensional arrays are supported.

The following program demonstrates the use of shape constructors:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable a shape [3] int,
    cardinalityvariable i,
    globalvariable sum int 0
]
statements [
    assign a shape [3] int [1,2,3],
    for [i=0:3] statements [
        assign sum (sum,+,(a,[i]))
    ],
    println [vnusout,sum]
]
```

It will produce the following output:

```
6
```

Shapes can only be cast to equivalent shape types; the size of the shape that is specified in the type will be ignored.

Internally, a shape maintains a block of memory to store the elements of the shape. For shapes that are expected to grow, it is often worthwhile to let the shape have a larger block of memory than is necessary for the current size. To support this, Vnus allows the manipulation of the *room* of an array. The room of an array is simply the number of shape elements that can be stored in the current memory block. The room of a shape can be changed with the `setroom` statement, which has the following syntax:

statement:
    `setroom` *shape-location room-expression*

After execution of this statement, it is ensured that the given shape has room for at least the given number of elements. The shape location expression must refer to a shape, the room expression must be of type `int`.

The `setroom` statement can only increase the room of a shape. To reduce the room of a shape, the `fitroom` statement must be used. It has the following syntax:

statement:
    `fitroom` *shape-location*

The statement reduces the room of the shape, so that it is sufficient to store a shape of the current size.

It is only guaranteed that the `setroom` and `fitroom` statements set the room of a shape to *at least* the specified amount. Even immediately after a `fitroom` statement, the room of a shape may be larger than necessary.

The room of a shape with unspecified size is unspecified.

The current room of a shape can be retrieved with the `getroom` operator, which has the following syntax:

expression:
    `getroom` *shape-expression*

4.2 Records

A record consists of a list of *fields*. For each field a *type* and a *field name* are specified. The order of the fields is significant.

Records can have an arbitrary number of fields (including zero), and can have arbitrary field names, with the restriction that all field names within one record type must be distinct. There is no relation between the field names of different record types. As a consequence, the same field name can be used in different record types.

The syntax of a record type is as follows:

type:
  `record [` $field_{list}$ `]`

field:
  identifier `:` *type*

With the syntax above, the expression of a record type must be repeated at every place it is used. To prevent the excessively verbose source files this would cause, and to allow recursive type references, Vnus also allows the declaration of named record types. Such a declaration has the following syntax:

declaration:
  `record` *identifier* `[` *fieldList* `]` $modifiers_{opt}$ $pragmas_{opt}$

These named records can then be used in type expressions with the following syntax:

type:
  `record` *identifier*

Record types are equivalent if they have the number of fields, and each pair of corresponding fields is equivalent. Fields are equivalent if they have equivalent types; the names of the field is not important for equivalence. Records of equivalent types can be assigned to eachother.

Record names are merely an abbreviation of a list of fields, they are always compared to other types as if the reference to the record name has been expanded.

A record type can be cast to another record type, provided that the target type has the same number of fields or less, and provided that the fields of the target type are equivalent to the first fields of the source type. No other casts are allowed.

An expression of a record type can be constructed using the `record` constructor expression, which has the following syntax:

expression:
  `record [` $expression_{list}$ `]`

It is possible to refer to individual fields of a record type in expressions and locations, using the following syntax:

location:
  `field` *record-expression field-identifier*

expression:
  `field` *record-expression field-identifier*

The following program demonstrates the use of records:

```
program
declarations [
    externalvariable vnusout int,
    record reca [a:int, b:int, c:char],
    globalvariable a record reca,
```

```
    globalvariable b record [a:int, b:int]
]
statements [
    assign a record [1,2,'x'],
    assign field b a 42,
    assign field b b -1,
    println [vnusout,"a:",field a a, " ", field a b, " ", field a c],
    println [vnusout,"b:",field b a, " ", field b b],
    assign b cast record [a:int, b:int] a,
    println [vnusout,"b:",field b a, " ", field b b]
]
```

It will produce the following output:

```
a:1 2 x
b:42 -1
b:1 2
```

4.3 Pointers

Pointers represent addresses. They are used to refer to nameless variables that are created by dynamic allocation, to implement 'reference' parameters, and to implement references to functions and procedures, in particular the virtual function tables that are necessary to implement virtual functions in object-oriented languages. Pointers may point to any type, including other pointers. They may also point to functions or procedures.

Pointer types are written as follows:

type:
  `pointer` *type*
  `procedure [` $type_{list}$ `]`
  `function [` $type_{list}$ `]` *type*

Function and procedure types are only used as the target of pointer types; function and procedure variables are not allowed.

In general, pointers may be cast to pointers to equivalent types.

For pointers to records the restrictions are more relaxed. This was done so that, for example, the pointer to the record of a given class can be cast to a pointer to the record of a superclass or subclass. For this reason a pointer to a record A may be cast to a pointer a record B, provided that casting from record A to record B is allowed, *or* casting from record B to record A is allowed.

Similarly, a pointer to a routine of type A may be cast to a routine of type B provided that all of the following conditions are met:

- The length of the parameter lists in types A and B are the same.
- Every the parameter type in type A can be cast to its corresponding parameter type in type B *or* every the parameter type in type B can be cast to its corresponding parameter type in type A.
- The return type of A may be cast to the return type of B.

The element pointed to by a pointer can be used as location or expression using the following syntax:

location:
  * *expression*

expression:
  * *expression*

The address of a variable, function, or procedure can be determined with the *address* expression, which is written as follows.

expression:
  `&` *location*

Apart from an address, a pointer can contain the special value `null`. It is guaranteed that this value is distinct from all valid addresses.

expression:
  `null`

The following program demonstrates the use of pointers:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable a int,
    globalvariable p pointer int
]
statements [
    assign a 0,
    assign p &a,
    assign *p 32,
    println [vnusout,a]
]
```

It will produce the following output:

```
32
```

For dynamic memory management, Vnus provides a `new` expression to create new variables, a `delete` statement to hint to the memory management system that a variable may be deleted, and a `garbagecollect` statement to force garbage collection.

After a memory block has been deleted, it cannot be used any more, any further reference of to the memory block results in unspecified behavior of the program.

Their syntax is as follows:

expression:
  `new` *type*
  `fillednew` *type init-expression*

statement:
  `delete` *location*
  `garbagecollect`

The `fillednew` version of the expression is only allowed for shape types; it fills all the elements of the new shape with the given value.

The following program demonstrates the use of `new` and `delete`:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable p pointer shape [dontcare] int
]
statements [
    assign p new shape [20] int,
    if (p,<>,cast pointer shape [dontcare] int null) statements [
        assign (*p,[0]) 122,
```

```
        assign (*p,[1]) -2,
        println [vnusout,(*p,[1])," ",(*p,[0])],
        delete p
    ]
    statements [
        println [vnusout,"Cannot allocate shape"]
    ]
]
```

It will produce the following output:

```
-2 122
```

Chapter 5

Parallelization constructs

Vnus provides support for a number of different *parallel execution models*. These execution models are:

- Data parallel. The language constructs for this execution model, `foreach` and the distributions of shapes, are described in chapter 5.2.
- SPC parallel. See [5] for a description of the SPC programming model. The language constructs for this execution model, `each` and `foreach`, are described in Section 5.3.
- Machine-level parallel. The language constructs for this execution model, `forkall` and the communication statements, are described in chapter 5.4.

The first two execution models cannot be translated directly onto parallel machines. Instead, they must be transformed by parallelization engines into Vnus programs that use the machine-level parallel constructs of Vnus.

Before the support for each programming model is described, the specification and numbering of processor arrays is described, since this is a more general issue.

5.1 Processor arrays

Vnus does not support explicit declaration of processor arrays. Instead, a multi-dimensional processor array is specified as a global pragma or as a command line parameter to the compiler. We consider this multi-dimensional array a 'view' on a one-dimensional processor array that contains all processors of the multi-dimensional array. These two processor arrays are the only ones that can be used in the program.

For example, the compiler can be invoked with

```
vnus -t5,4,12
```

This specifies a processor array of $5 \times 4 \times 12$ elements.

The size of the processor array is available within the program through the predefined array sizesProcessorArray. Thus, with the invocation shown above `sizesProcessorArray[0] = 5`, `sizesProcessorArray[1] = 4`, and `sizesProcessorArray[2] = 12`.

The variable numberOfProcessors contains the product of all elements of sizesProcessorArray; for example, when invoked with

```
vnus -t5,4,12
```

it will be 240. The variable can be interpreted as the size of the one-dimensional processor array.

5.1.1 The numbering of processors

Since the more-dimensional processor array is a view on the one-dimensional processor array, we must define a mapping between processor vectors that index the more-dimensional processor array, and processor numbers that index the one-dimensional processor array.

For this reason we define that every processor vector v maps to a processor number p as follows:

$$p = v_0 + v_1 \cdot p_0 + v_2 \cdot p_0 \cdot p_1 + \dots$$

Where p_n are elements of the array numberOfProcessors. This provides a one-to-one mapping between processor vectors and processor numbers.

For example, if we have a processor array of $5 \times 8 \times 12$, and we have a processor vector `v = [1,2,3]`, $p = 1 + 2 \cdot 5 + 3 \cdot 5 \cdot 8$ or: $p = 131$.

5.1.2 The representation of replication groups

Since in the data-parallel programming model we allow replicated dimensions, we sometimes want to specify ranges of processors. Suppose we have an array of $5 \times 3 \times 6$ processors, and we distribute a shape as follows:

```
shape [100,100,100] [cyclic,replicated,cyclic] of integer
```

Since the second dimension is replicated, every element is owned by a *set* of processors. For example, element `A[2,2,2]` is now owned by processors `[2,0,2]`, `[2,1,2]`, and `[2,2,2]`. Since we know that such a list always spans the entire range of processors in each replicated dimension, we can write this more conveniently as `[2,*,2]`, where `*` indicates the entire range of processors in that dimension.

Since we must sometimes manipulate such a set of owners, we must find a convenient way to represent it. Fortunately, we can take advantage of the special properties of owner sets: every owner set can be represented as a base processor vector and a vector of booleans that indicate whether a dimension is replicated. For example, vector `[2,*,2]` can be represented as `([2,0,2],[false,true,false])`.

The base processor vector can be represented as a processor number in the usual way, and the vector of booleans can be represented as a (binary) number, resulting in `(32,2)`. This, in turn, can be encoded by multiplying the latter number with numberOfProcessors and adding it to the processor number, resulting in $32 + 90 \cdot 2 = 212$.

By encoding the set of processors this way, every possible owner set can be encoded as a single number, and if the set contains only one processor (i.e. there is no replication), the encoding of the set is the processor number itself. The result of all this is that replication sets can be manipulated as if they were single processor numbers.

As a convenience for Vnus users, the predefined global variable `allProcessors` contains the encoding of the set of all processors.

5.2 Data-parallel constructs

The data-parallel extensions of Vnus allow the implicit specification of parallelism through *distribution specifications*. The user specifies the distribution of each array over one of the two supported processor arrays, and the compiler uses the distribution to determine the distribution of computation over the processors. This is done in a rather straightforward manner: every processor is responsible for the computations on the variables that it owns. This is called the *owner computes* rule.

Apart from the distribution specification, the data-parallel programming model requires the specification of statements that can be executed in parallel. For this purpose the `forall` statement is provided.

5.2.1 Distribution specifications

To support the data-parallel programming model, every Vnus shape can be given a *distribution* that specifies how it is distributed over the processors.

As was shown in chapter 4, shape type specifications have the following syntax:

type:
  `shape` *sizes* $distributions_{opt}$ *type*

At that point the syntax of *distributions* was left unspecified. Now we define it as follows:

distributions:
  `[` $distribution_{list}$ `]`

distribution:
  `block`
  `cyclic`
  `blockcyclic` *blocksize-expression*
  `replicated`
  `collapsed`
  `local` *processor-expression*

A shape must have a distribution specification for each of its dimensions. Thus, the size of the distribution list must be the same as the rank of the shape.

Possible distribution specifications for a shape dimension are:

- `blockcyclic n`. The elements in this shape dimension are distributed in groups of `n` elements over the processor dimension.
- `block`. Equivalent to `blockcyclic` $\lceil n_e/n_p \rceil$, where n_e is the number of elements in this shape dimension, and n_p is the number of processors in this processor dimension.
- `cyclic`. Equivalent to `blockcyclic` 1.
- `replicated`. All elements in this shape dimension are replicated on all processors in this dimension.
- `collapsed`. Skip this shape dimension.
- `local p`. All elements are stored on processor '`p`'.

Each distribution specification, except the `collapsed` specification, uses one processor dimension. The list of distributions of a shape must use either the entire multi-dimensional processor array, or only the one-dimensional processor array. In other words, in a program with a processor array of rank n, every distribution must have either 1 or n non-collapsed dimensions.

A shape without a distribution specification has an unspecified distribution. In other words, there is no default distribution.

The assignment of a shape overwrites the distribution of the destination shape. Consequently, it is pointless to specify the distribution of a formal shape parameter, since this distribution will always be overwritten by the distribution of the actual parameter.

Casting a shape causes the shape to assume the specified distribution, or, if the cast type does not contain a distribution specification, causes it to have an unspecified distribution.

Changing the size of a shape with the `setsize` statement does not change the distribution of the shape; it merely changes the size of the local part of the shape on each processor. In particular, a `block` distributed dimension may not be `block` distributed any more.

The following program demonstrates the use of distribution specifications:

```
program
declarations [
    globalvariable a shape [1000,1000] [block,replicated] int,
    globalvariable b shape [1000,1000] int,
```

```
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
    // 'a' is block-distributed over its first dimension, and
    // replicated over its second dimension. The distribution
    // assumes a two-dimensional (view on the) processor array.
    //
    // At this point, the distribution of 'b' is unspecified.
    for [i=0:1000,j=0:1000] statements [
        assign (a,[i,j]) 0
    ],
    assign b a,
    // Now 'b' has the same contents and distribution as 'a'.

    assign b cast shape [1000,1000] [blockcyclic 10,collapsed] int a
    // 'b' again has the same contents as 'a', but now it is
    // distributed in blocks of 10 elements over the
    // one-dimensional (view of the) processor array.
]
```

This program does not produce any output.

5.2.2 Distribution inquery functions

To inquire about the distribution of a shape, the following expressions are provided:

expression:
`ismultidimdist` *expression*
`getblocksize` *expression expression*
`sender` *location*
`owner` *location*
`isowner` *location expression*

Given a shape, `ismultidimdist` returns `true` if it is distributed over the multi-dimensional processor array, or `false` if it is distributed over the one-dimensional processor array.

Given a shape and a dimension number (an `int`), `getblocksize` returns an `int` representing the block size in that dimension. Only block, cyclic and blockcyclic distributions return a useful block size. For other distributions the returned value is unspecified.

Given a selection location, `sender` returns an `int` representing the processor that is responsible for sending this element. Note that only one processor will be returned, even if the element is owned by multiple processors.

Given a selection location, `owner` returns an `int` representing the processors that own this element. The set is encoded as described in section 5.1.

Given a selection location and an `int` representing a processor number, `isowner` returns `true` if the given processor owns that element, or `false` if it does not.

5.2.3 The `forall` statement

The `forall` statement executes the given body for every element in the iteration space, and afterwards merges the state changes caused by the iterations into a new program state. Since the state changes are only merged afterwards, every iteration only sees the state at the start of the `forall` statement, plus the state changes it causes itself. If a state change of an iteration conflicts with a state change of another iteration, the resulting state is unspecified.

statement:
`forall` *cardinalities block*

The following program demonstrates the use of the `forall` statement:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable a shape [10] int,
    cardinalityvariable i,
    cardinalityvariable j,
    cardinalityvariable k
]
statements [
    forall [i=0:getsize a 0] statements [
        assign (a,[i]) i
    ],
    for [k=0:getsize a 0] statements [
        print [vnusout,(a,[k])," "]
    ],
    println [vnusout],
    forall [j=0:(getsize a 0,-,1)] statements [
        assign (a,[j]) (a,[(j,+,1)])
    ],
    for [k=0:getsize a 0] statements [
        print [vnusout,(a,[k])," "]
    ],
    println [vnusout]
]
```

This program produces the following output:

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 9
```

5.3 SPC parallel constructs

Since Vnus is designed as intermediate language for Spar, the language supports the Spar `each` and `foreach` statements. Without intervention, the Vnus backend will produce sequential code for these statements. For more efficient execution they must be transformed to a simpler form.

5.3.1 The `each` statement

The `each` statement is written as follows:

> *statement*:
> `each [` $statement_{list}$ `]`

The `each` statement executes the statements in the given list in an unspecified order. It is guaranteed, even for compound statements, that every statement is executed as one state transition.

Given a `each` statement:

```
each [ s1, s2 ]
```

the compiler will choose one of the execution orders `statements [ s1, s2 ]`, or `statements [ s2, s1 ]`.

5.3.2 The `foreach` statement

The `foreach` statement is a parameterized version of the `each` statement of the previous section. It is written as follows:

statement:
`foreach` *cardinalities block*

The body of the `foreach` is executed for every element in the iteration space. The order of execution is unspecified, but it is guaranteed, even for compound statements, that every iteration is executed as one state transition. Thus, iterations cannot influence each other during their execution.

The following program demonstrates the use of the `foreach`:

```
program
declarations [
    externalvariable vnusout int,
    globalvariable a shape [100] int,
    globalvariable sum int 0,
    cardinalityvariable i,
    cardinalityvariable j
]
statements [
    foreach [i=0:getsize a 0] statements [
        assign (a,[i]) i
    ],
    foreach [j=0:getsize a 0] statements [
        assign sum (sum,+,(a,[j]))
    ],
    println [vnusout,sum]
]
```

This program produces the following output:

```
4950
```

5.3.3 The `volatile` modifier

If a variable is declared `volatile`, its value may be changed asynchronously by other execution threads. This allows parallelizer engines to ignore this variable in resource contention analysis.

The Vnus compiler ignores the `volatile` modifier; it is only used by code transformation engines.

5.4 Machine-level parallel constructs

The language constructs that were described in the previous sections are not easy to map onto the parallelization constructs that are usually available on parallel computer systems. Instead, these constructs are translated to machine-level parallelization constructs by separate code transformation engines. These machine-level constructs are described in this section.

The machine-level parallel constructs assume a distributed-memory machine, where the processors communicate through message passing. Without modification to Vnus it is possible to define a shared-memory machine model, but the current Vnus compiler does not support this.

5.4.1 The `fork` statement

statement:
`fork` *cardinalities* `[` $statement_{list}$ `]`

The `fork` statement executes the statements in the given list in an unspecified order. All statements are executed completely independently.

The current implementation the Vnus backends do not support this statement, so it must always be translated away by optimizer engines.

5.4.2 The `forkall` statement

statement:
`forkall` *cardinalities block*

The body of the `forkall` is executed for every element in the iteration space. All iterations are executed completely independently.

The current Vnus backend only supports the `forkall` statement in a very specific context: the `forkall` statement must be the sole contents of the top-level statement block of a Vnus program. Moreover, the `forkall` must have a one-dimensional iteration range. In this form, the `forkall` can be interpreted as the specification of an SPMD (single program, multiple data) program. The cardinality variable specifies the number of the processor to execute the body instance on (that is, it must be `numberOfProcessors`), and the body of the `forkall` specifies the SPMD program to execute.

5.4.3 Single-element communication

The different execution threads of an SPMD program exchange information through messages. For that purpose a `send` statement are provided, with the following syntax:

statement:
`send` *me-expression dest-expression payload-expression*

In this order the parameters are: my processor number, the processor to send to, and the element to send. The processor numbers must be of type `int`, and the payload element must be of a base type, but not of type `string`.

Upon completion of the `send` statement, the element may not yet have reached its destination.

A `receive` statement is also provided:

statement:
`receive` *me-expression src-expression location*

In this order the parameters are: my processor number, the processor to receive from, and the location to receive the element in. The processor numbers must be of type `int`, and the element to receive in must be of a base type but not of type `string`.

Upon completion of the `receive` statement, the data is guaranteed to be available. If necessary, execution is suspended until the data has been received.

5.4.4 Block communication

Since single-element communication is highly inefficient on most systems, Vnus also provides send and receive statements that work on blocks of data. The block send statement has the following syntax:

statement:
`blocksend` *me-expression dest-expression buf-expression sz-expression*

In this order the parameters are: my processor number, the processor to send to, the shape to send, and the number of elements to send. The processor numbers and the number of elements to send must be of type `int`, and the shape to send from must be a one-dimensional shape of a base type but not of type `string`.

Upon completion of the `blocksend` statement, the elements may not yet have reached their destination.

And the block receive statement has the following syntax:

statement:
`blockreceive` *me-expression src-expression buf-location sz-location*

In this order the parameters are: my processor number, the processor to receive from, the location of the shape to put the elements in, and the location to put number of elements that were received in. The processor numbers and the number of elements received must be of type `int`, and the shape to receive in must be a one-dimensional shape of a base type but not of type `string`.

Upon completion of the `receive` statement, the data is guaranteed to be available. If necessary, execution is suspended until all the data has been received.

5.4.5 The `barrier` statement

In some cases it is useful to pause execution until all processors have reached the same execution point. For that purpose the `barrier` statement is provided.

statement:
  `barrier`

5.4.6 `local` annotations

The `local` annotation specifies that every processor in a SPMD program has a local copy of the given variable, that does not have to be kept synchronized with the copies of the variable on other processors.

Appendix A

The Vnus grammar

This appendix specifies the grammar of Vnus.

A.1 Program

program:
`program` *globalPragmas*$_{opt}$ *declarations block*

A.2 Declarations

declarations:
`declarations [` *declaration*$_{list}$ `]`

declaration:
routineDeclaration
variableDeclaration
typeDeclaration

routineDeclaration:
functionDeclaration
procedureDeclaration
externalFunctionDeclaration
externalProcedureDeclaration

variableDeclaration:
globalVariableDeclaration
externalVariableDeclaration
cardinalityVariableDeclaration
localVariableDeclaration
formalVariableDeclaration

typeDeclaration:
recordDeclaration

globalVariableDeclaration:
`globalvariable` *identifier type expression*$_{opt}$ *modifiers*$_{opt}$ *pragmas*$_{opt}$

functionDeclaration:
`function` *identifier formalParameters identifier type modifiers*$_{opt}$ *pragmas*$_{opt}$ *block*

procedureDeclaration:
`procedure` *identifier formalParameters* $modifiers_{opt}$ $pragmas_{opt}$ *block*

localVariableDeclaration:
`localvariable` *identifier scopename type* $expression_{opt}$ $modifiers_{opt}$ $pragmas_{opt}$

formalVariableDeclaration:
`formalvariable` *identifier scopename type* $modifiers_{opt}$ $pragmas_{opt}$

cardinalityVariableDeclaration:
`cardinalityvariable` *identifier* $modifiers_{opt}$ $pragmas_{opt}$

externalVariableDeclaration:
`externalvariable` *identifier type* $modifiers_{opt}$ $pragmas_{opt}$

externalFunctionDeclaration:
`externalfunction` *identifier formalParameters type* $modifiers_{opt}$ $pragmas_{opt}$

externalProcedureDeclaration:
`externalprocedure` *identifier formalParameters* $modifiers_{opt}$ $pragmas_{opt}$

recordDeclaration:
`record` *identifier* `[` *fieldList* `]` $modifiers_{opt}$ $pragmas_{opt}$

modifiers:
modifier
modifiers modifier

modifier: one of
`const local unchecked volatile`

block:
`statements` $scopename_{opt}$ $pragmas_{opt}$ `[` $labeledStatement_{list}$ `]`

labeledStatement:
$label_{opt}$ $pragmas_{opt}$ *statement*

statement:
imperative
control
parallelization
communication
memoryManagement
support

formalParameters:
`[` $formalParameter_{list}$ `]`

formalParameter:
identifier

A.3 Flow of Control

control:
while
dowhile
for
if

block
switch
return
valueReturn
goto
throw
rethrow
catch

while:
`while` *expression block*

dowhile:
`dowhile` *expression block*

for:
`for` *cardinalities block*

if:
`if` *expression block block*

switch:
`switch` *expression* `[` $switchCase_{list}$ `]`

return:
`return`

valueReturn:
`return` *expression*

goto:
`goto` *labelName*

throw:
`throw` *expression*

rethrow:
`rethrow`

catch:
`catch` *formalParameter block block*

switchCase:
`(` *intLiteral* `,` *block* `)`
`(` `default` `,` *block* `)`

cardinalities:
`[` $cardinality_{list}$ `]`

cardinality:
identifier `:` *expression*

A.4 Parallelization

parallelization:
forall

forkall
fork
foreach
each

forall:
`forall` *cardinalities block*

forkall:
`forkall` *cardinalities block*

fork:
`fork [` $statement_{list}$ `]`

foreach:
`foreach` *cardinalities block*

each:
`each [` $statement_{list}$ `]`

A.5 Communication

communication:
barrier
signal
wait
send
receive
blockSend
blockReceive

barrier:
`barrier`

wait:
`wait` *expression*

signal:
`signal` *expression*

send:
`send` *expression expression expression*

receive:
`receive` *expression expression location*

blockSend:
`blocksend` *expression expression expression expression*

blockReceive:
`blockreceive` *expression expression location location*

A.6 Imperative

imperative:
assignment
procedureCall
setSize
setRoom
fitRoom

assignment:
`assign` *location expression*

procedureCall:
`procedurecall` *routineExpression actualParameters*

setSize:
`setsize` *location sizes*

setRoom:
`setroom` *location expression*

fitRoom:
`fitroom` *location*

routineExpression:
identifier
`*` *expression*

A.7 Memory management

memoryManagement:
delete
garbageCollect

delete:
`delete` *expression*

garbageCollect:
`garbagecollect`

A.8 Support statements

support:
empty
print
println

empty:
`empty`

print:
`print` *actualParameters*

println:
`println` *actualParameters*

A.9 Expressions and locations

location:
identifier
`field` *expression identifier*
selectionLocation
`expressionpragma` *pragmas location*
`*` *expression*
`where` *scopename location*

selectionLocation:
  (*expression* , *selectors*)

expression:
  literalExpression
  accessExpression
  constructorExpression
  operatorExpression
  shapeInfoExpression
  miscellaneousExpression

literalExpression:
  byteLiteral
  shortLiteral
  intLiteral
  longLiteral
  floatLiteral
  doubleLiteral
  charLiteral
  booleanLiteral
  stringLiteral
  nullLiteral

accessExpression:
  identifier
  (*expression* , *selectors*)
  `field` *expression identifier*
  * *expression*
  `functioncall` *routineExpression actualParameters*

constructorExpression:
  `complex` *expression expression*
  `shape` *sizes type* [$expression_{list}$]
  `record` [$expression_{list}$]
  & *location*
  `new` *type*
  `fillednew` *type expression*

operatorExpression:
  `cast` *type expression*
  `if` *expression expression expression*
  `where` *scopename expression*
  (*unaryOperator* , *expression*)
  (*expression* , *binaryOperator* , *expression*)

shapeInfoExpression:
  `ismultidimdist` *expression*
  `getblocksize` *expression expression*
  `getsize` *expression expression*
  `getlength` *expression*
  `getroom` *expression*
  `sender` *location*
  `owner` *location*
  `isowner` *location expression*

miscellaneousExpression:
  `expressionpragma` *pragmas expression*

`israised` *expression*

selectors:
`[` $selector_{list}$ `]`

selector:
expression

actualParameters:
`[` $actualParameter_{list}$ `]`

actualParameter:
expression

distributions:
`[` $distribution_{list}$ `]`

distribution:
`dontcare`
`block`
`cyclic`
`blockcyclic` *expression*
`replicated`
`collapsed`
`local` *expression*

type:
baseType
`shape` *sizes* $distributions_{opt}$ *type*
`record [` $field_{list}$ `] pointer` *type*
`procedure [` $type_{list}$ `]`
`function [` $type_{list}$ `]` *type*

baseType: one of
`string boolean byte short int long float double char complex`

field:
identifier `:` *type*

sizes:
`[` $size_{list}$ `]`

size:
`dontcare`
expression

A.10 Miscellaneous

scopename:
identifier

label:
labelName `:`

labelName:
identifier

globalPragmas:
`pragma` *pragmas*

pragmas:
`[` $pragma_{list}$ `]`

pragma:
identifier
identifier `=` *PragmaExpression*

PragmaExpression:
LiteralPragmaExpression
NamePragmaExpression
ListPragmaExpression

LiteralPragmaExpression:
intLiteral
floatLiteral
doubleLiteral
stringLiteral
booleanLiteral

NamePragmaExpression:
identifier
`@` *identifier*

ListPragmaExpression:
`(` *PragmaExpressions* `)`

PragmaExpressions:
empty
PragmaExpressions PragmaExpression

A.11 Tokens

unaryOperator: one of
`not + ~`

binaryOperator: one of
`* + - / < << <= <> = > >= >> >>> and mod or xor`

booleanLiteral: one of
`true false`

nullLiteral:
`null`

keyword: one of
`and assignment barrier block blockcyclic blockreceive blocksend boolean byte cardinalityvariable catch collapsed complex cyclic declarations delete dontcare double each empty expression expressionpragma expressions externalfunction externalprocedure externalvariable false field fillednew fitroom flag float forall foreach fork forkall formalvariable function`

`functioncall garbagecollect getblocksize getlength getsize globalvariable goto if integer isowner iteration local localvariable long mod new not null or owner pointer pragma print println procedure procedurecall program receive replicated rethrow return send sender setroom setsize shape short statements string switch throw true value while xor`

A.12 Lexical entities

In the productions in this section, whitespace is significant.

byteLiteral:
  intNumber byteSuffix

byteSuffix: one of
  `b B`

shortLiteral:
  intNumber shortSuffix

shortSuffix: one of
  `s S`

intLiteral:
  intNumber
  intNumber intSuffix

intSuffix: one of
  `i I`

longLiteral:
  intNumber longSuffix

longSuffix: one of
  `l L`

floatLiteral:
  floatNumber
  floatNumber floatSuffix

floatSuffix: one of
  `f F`

doubleLiteral:
  floatNumber doubleSuffix

doubleSuffix: one of
  `d D`

charLiteral:
  ' *char* '

stringLiteral:
  " *stringChars* "

stringChars:
  stringChar
  stringChars stringChar

stringChar:
  any *char* except an unescaped '`"`'

identifier:
  identifierChars but not a *keyword*

identifierChars:
  letter
  identifierChars letter
  identifierChars digit

letter: one of
  `_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J`
  `K L M N O P Q R S T U V W X Y Z`

floatNumber:
  $sign_{opt}$ *digits floatExponent*
  $sign_{opt}$ `.` *digits* $floatExponent_{opt}$
  $sign_{opt}$ *digits* `.` $digits_{opt}$ $floatExponent_{opt}$

floatExponent:
  exponentIndicator $sign_{opt}$ *digits*

exponentIndicator: one of
  `e E`

intNumber:
  $sign_{opt}$ *digits*

digits:
  digit
  digits digit

digit: one of
  `0 1 2 3 4 5 6 7 8 9`

sign: one of
  `+ -`

char:
  any character
  escapedCharacter

escapedCharacter:
  `\b`
  `\f`
  `\n`
  `\r`
  `\t`
  `\"`
  `\\`
  `\` *digit*
  `\` *digit digit*
  `\` *digit digit digit*

Appendix B

All known supported pragmas

This appendix describes the pragmas that are supported by one or more of the currently known engines working on Vnus.

B.1 The `boundscheck` pragma

The `boundscheck` boolean pragma annotates a selection *expression* or *location*, and the `owner`, `isowner`, and `sender` expression. If set to `false`, code is generated for that expression or location that does no bounds checking on the selection expression or location. If set to `true`, code is generated that *does* do bounds checking. Without the pragma, no explicit choice is made, but in the current implementation bounds checking is done.

This pragma can also be used as a global pragma, in which case it is considered to annotate all array accesses, except for those that have their own `boundscheck` pragma.

B.2 The `processors` pragma

As a global pragma the `processors` pragma is recognized by the backend.

When a program is annotated with this pragma, it will use the processor array specified here. Any processor array that has been specified on the command line will be ignored.

A string is used to specify the processor array since there is no concept of an array constant in Vnus. The string should consist of a list of numbers, separated by arbitrary other characters. Thus, strings like `3,4,5`, and `3x4x12` and even `a processor array of 3 by 4 by 23` are valid processor array specifications.

B.3 The `location` pragma

The `location` pragma annotates formal parameters, and indicates that the annotated formal parameter always has actual parameter values that can be interpreted as locations.

B.4 The `readonly` pragma

The `readonly` pragma annotates formal parameters, and indicates that the annotated formal parameter is never modified.

Appendix C

Pragmas supported by the Vnus to C translator

This appendix describes the pragmas that are supported by the Vnus to C translator.

C.1 The `boundscheck` pragma

The `boundscheck` boolean pragma annotates a selection *expression* or *location*, and the `owner`, `isowner`, and `sender` expression. If set to `false`, code is generated for that expression or location that does no bounds checking on the selection expression or location. If set to `true`, code is generated that *does* do bounds checking. Without the pragma, no explicit choice is made, but in the current implementation bounds checking is done.

This pragma can also be used as a global pragma, in which case it is considered to annotate all array accesses, except for those that have their own `boundscheck` pragma.

C.2 The `processors` pragma

As a global pragma the `processors` pragma is recognized by the backend.

When a program is annotated with this pragma, it will use the processor array specified here. Any processor array that has been specified on the command line will be ignored.

A string is used to specify the processor array since there is no concept of an array constant in Vnus. The string should consist of a list of numbers, separated by arbitrary other characters. Thus, strings like `3,4,5`, and `3x4x12` and even `a processor array of 3 by 4 by 23` are valid processor array specifications.

C.3 The `location` pragma

The `location` pragma annotates formal parameters, and indicates that the annotated formal parameter always has actual parameter values that can be interpreted as locations.

C.4 The `readonly` pragma

The `readonly` pragma annotates formal parameters, and indicates that the annotated formal parameter is never modified.

Appendix D

The Tm data structure definitions for Vnus

This appendix lists the file `vnus.ds`, which defines Vnus in terms of Tm data structures.

```
|| File: vnus.ds
||
|| Datastructures describing the intermediate programming language Vnus

|| An origin descriptor.
|| This is tacked on to all sorts of constructs for which it is convenient
|| to track the origin.
origin == ( file:filename, line:int );

|| A symbol with its origin.
origsymbol == ( sym:tmsymbol, orig:origin );

|| The aliases below are used to distinguish between the name spaces the
|| various symbols are in.
varname -> origsymbol;
pragmaname -> origsymbol;
labelname -> origsymbol;
scopename -> tmsymbol;
filename -> tmsymbol;
fieldname -> origsymbol;

vnusprog == ( pragmas:[Pragma], declarations:[declaration], statements:block );

declaration ~= { name:varname, flags:int, pragmas:[Pragma] };

RoutineDeclaration = declaration +
    DeclFunction: { parms:[varname], rettype:type, body:block } |
    DeclProcedure: { parms:[varname], body:block } |
    DeclExternalFunction: { parms:[varname], rettype:type } |
    DeclExternalProcedure: { parms:[varname] }
    ;

VariableDeclaration = declaration +
    DeclGlobalVariable: { t:type, init:optexpression } |
    DeclLocalVariable: { scope:scopename, t:type, init:optexpression } |
    DeclFormalVariable: { scope:scopename, t:type } |
    DeclCardinalityVariable: {} |
```

```
    DeclExternalVariable: { t:type }
    ;

TypeDeclaration = declaration +
    DeclRecord: { fields:[field] }
    ;

|| A statement block with a scope name
block == ( scope:scopename, orig:origin, pragmas:[Pragma], on:[OwnerExpr], statements:[statement] );

switchCase = { body:block } +
    SwitchCaseValue: { cond:vnus_int } |
    SwitchCaseDefault: {}
    ;

statement ~= { label:labelname, orig:origin, pragmas:[Pragma], on:[OwnerExpr] };

Imperative = statement +
    SmtAssign: { lhs:location, rhs:expression } |
    SmtProcedureCall: { proc:expression, parameters:[expression] } |
    SmtExpression: { x:expression } |
    SmtSetSize: { shape:location, sizes:[size] } |
    SmtSetRoom: { shape:location, sz:expression } |
    SmtFitRoom: { shape:location }
    ;

Control = statement +
    SmtWhile: { cond:expression, body:block } |
    SmtDoWhile: { cond:expression, body:block } |
    SmtFor: { cards:[cardinality], body:block } |
    SmtIf: { cond:expression, thenbody:block, elsebody:block } |
    SmtBlock: { body:block } |
    SmtSwitch: { cond:expression, cases:[switchCase] } |
    SmtReturn: {} |
    SmtValueReturn: { v:expression } |
    SmtGoto: { target:labelname } |
    SmtThrow: { elm:expression } |
    SmtRethrow: {} |
    SmtCatch: { elm:varname, body:block, handler:block }
    ;

Parallelization = statement +
    SmtForall: { cards:[cardinality], map:[OwnerExpr], body:block } |
    SmtForkall: { cards:[cardinality], map:[OwnerExpr], body:block } |
    SmtFork: { statements:[statement] } |
    SmtForeach: { cards:[cardinality], map:[OwnerExpr], body:block } |
    SmtEach: { statements:[statement] }
    ;

Semaphores = statement +
    SmtSignal: { var:expression } |
    SmtWait: { var:expression }
    ;

Communication = statement +
    SmtBarrier: {} |
```

```
    SmtSend: { me:expression, dest:expression, elm:expression } |
    SmtReceive: { me:expression, src:expression, elm:location } |
    SmtBlocksend: {
        me:expression,
        dest:expression,
        elm:expression,
        nelm:expression
    } |
    SmtBlockreceive: {
        me:expression,
        src:expression,
        elm:location,
        nelm:location
    }
    ;

MemoryManagement = statement +
    SmtDelete: { adr:expression } |
    SmtGarbageCollect: {}
    ;

Support = statement +
    SmtEmpty: {} |
    SmtPrint: { argv:[expression] } |
    SmtPrintLn: { argv:[expression] }
    ;

size =
    SizeDontcare: { orig:origin } |
    SizeExpression: { x:expression };

distribution =
    DistDontcare: {} |
    DistBlock: {} |
    DistCyclic: {} |
    DistBC: { blocksize:expression } |
    DistReplicated: {} |
    DistCollapsed: {} |
    DistLocal: { proc:expression }
    ;

|| A type
type =
    TypeMap: { map:OwnerExpr, t:type } |
    TypeBase: { base:BASETYPE } |
    TypeShape: { sizes:[size], distr:[distribution], elmtype:type } |
    TypePointer: { elmtype:type } |
    TypeRecord: { fields:[field] } |
    TypeNamedRecord: { name:varname } |
    TypeProcedure: { formals:[type] } |
    TypeFunction: { formals:[type], rettype:type } |
    TypePragmas: { pragmas:[Pragma], t:type }
    ;

field == ( name:fieldname, elmtype:type );
```

```
|| A cardinality - a variable that iterates over a given range, with
|| a given stride. Contrary to many other range notations, the stride
|| should not be negative.
cardinality == (
    name:varname,
    lowerbound:expression,
    upperbound:expression,
    stride:expression,
    secondaries:[secondary]
);

|| A secondary - a variable that follows a cardinality
secondary == (
    name:varname,
    lowerbound:expression,
    stride:expression
);

|| A Vnus location expression
location =
    LocName: { name:varname } |
    LocField: { rec:expression, field:fieldname } |
    LocFieldNumber: { rec:expression, field:vnus_int } |
    LocSelection: { shape:expression, indices:[expression] } |
    LocPragma: { pragmas:[Pragma], l:location } |
    LocOwner: { on:[OwnerExpr], l:location } |
    LocDeref: { ref:expression } |
    LocWhere: { scope:scopename, l:location }
    ;

AssertLocation = location + { exception:expression } +
    LocNotNullAssert: { l:location }
    ;

optexpression =
    OptExprNone: {} |
    OptExpr: { x:expression }
    ;

|| A Vnus expression.
expression ~= {};

LiteralExpression = expression +
    ExprByte: { v:vnus_byte } |
    ExprShort: { v:vnus_short } |
    ExprInt: { v:vnus_int } |
    ExprLong: { v:vnus_long } |
    ExprFloat: { v:vnus_float } |
    ExprDouble: { v:vnus_double } |
    ExprChar: { c:vnus_char } |
    ExprBoolean: { b:vnus_boolean } |
    ExprString: { s:vnus_string } |
    ExprNull: {}
    ;

AccessExpression = expression +
```

```
    ExprName: { name:varname } |
    ExprSelection: { shape:expression, indices:[expression] } |
    ExprField: { rec:expression, field:fieldname } |
    ExprFieldNumber: { rec:expression, field:vnus_int } |
    ExprDeref: { ref:expression } |
    ExprFunctionCall: { function:expression, parameters:[expression] }
    ;

ConstructorExpression = expression +
    ExprComplex: { re:expression, im:expression } |
    ExprShape: { sizes:[size], elmtype:type, arr:expression, nelms:uint } |
    ExprRecord: { fields:[expression] } |
    ExprAddress: { adr:location } |
    ExprNew: { t:type } |
    ExprFilledNew: { t:type, init:expression }
    ;

OperatorExpression = expression +
    ExprCast: { t:type, x:expression } |
    ExprIf: { cond:expression, thenval:expression, elseval:expression } |
    ExprWhere: { scope:scopename, x:expression } |
    ExprUnop: { optor:UNOP, operand:expression } |
    ExprBinop: { operanda:expression, optor:BINOP, operandb:expression }
    ;

ShapeInfoExpression = expression +
    ExprIsMultidimDist: { shape:expression } |
    ExprGetBlocksize: { shape:expression, dim:expression } |
    ExprGetSize: { shape:expression, dim:expression } |
    ExprGetLength: { shape:expression } |
    ExprGetRoom: { shape:expression } |
    ExprSender: { shape:location } |
    ExprOwner: { shape:location } |
    ExprIsOwner: { shape:location, proc:expression }
    ;

AssertExpression = expression + { exception:expression } +
    ExprNotNullAssert: { x:expression }
    ;

MiscellaneousExpression ~= expression +
    ExprPragma: { pragmas:[Pragma], x:expression } |
    ExprNewOwner: { on:[OwnerExpr], x:expression } |
    ExprIsRaised: { var:expression } |
    || Only used internally for the list of expressions of a subscript
    || and a shape expression
    ExprArray: { elms:[expression] }
    ;

Pragma = { name:pragmaname } +
    FlagPragma: {} |                || Being there is enough
    ValuePragma: { x:PragmaExpression }
    ;

PragmaExpression ~= {};
```

```
|| Various types of literals
LiteralPragmaExpression = PragmaExpression +
    NumberPragmaExpression: { v:vnus_double } |
    StringPragmaExpression: { s:vnus_string } |
    BooleanPragmaExpression: { b:vnus_boolean }
    ;

SymbolPragmaExpression = PragmaExpression +
    NamePragmaExpression: { name:origsymbol } |
    ExternalNamePragmaExpression: { name:varname }
    ;

ListPragmaExpression = PragmaExpression + { l:[PragmaExpression] };

OwnerExpr =
    NewOwnDontCare: {} |
    NewOwnReplicated: {} |
    NewOwnLocation: { proc:expression, distr:[DistrExpr] } |
    NewOwnMap: { parms:[varname], on:OwnerExpr }
    ;

|| intexpressions???
DistrExpr =
    NewDistDontcare: {} |
    NewDistBlock: { index:expression, size:expression } |
    NewDistCyclic: { index:expression, extent:expression } |
    NewDistBC: { index:expression, size:expression, extent:expression } |
    NewDistReplicated: {} |
    NewDistLocal: { index:expression } |
    OwnerOf: { with:location }
    ;

|| Local Variables:
|| tab-width:8
|| End:
```

Appendix E

Unresolved issues

- Special conversion routines such as `Double.longBitsToDouble` should be provided.
- The Unicode character set should be supported, but this will require some work.
- Overflow semantics of Java?
- Doing a `setsize` on a block-distributed shape causes it to be no longer block-distributed.
- Currently, the `foreach` has no mask.
- There should be an `exit` statement, but that's difficult to implement in a parallel system.

Bibliography

[1] P.F.G. Dechering. The denotational semantics of booster, a working paper, version 2.0. Technical report CP-95-05, Delft University of Technology, 1996.

[2] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, August 1996.

[3] E.M. Paalvast. *Programming for Parallelism and Compiling for Efficiency*. PhD thesis, Delft University of Technology, June 1992.

[4] C. van Reeuwijk. Tm users' manual. Technical report, Delft University of Technology, Department of Electrical Engineering, Delft, The Netherlands, 1992.

[5] Arjan J.C. van Gemund. Compile-time optimization and the SPC parallel programming model. In Michael Gerndt, editor, *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, pages 45–56. KFA, December 1996.

[6] Kees van Reeuwijk. WWW page, 1996. `http://www.pds.twi.tudelft.nl/projects/tm`.

Index