Delft University of Technology
Parallel and Distributed Systems Report Series

Spar language specification

Containing a description of the Timber compiler version 1.5

C. van Reeuwijk
Delft University of Technology

report number PDS-2001-003

PDS

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.twi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://pds.twi.tudelft.nl/

Contents

Preface

This document describes Spar/Java language, a superset of Java. The document also describes the capabilities and restrictions of version 1.5 of the Timber compiler, which implements Spar/Java language.

Since Spar/Java is derived from Java, I have chosen not to repeat the Java language specification here, but to refer to the official Java language specification [7] intensively. Any suggestions for improvement are welcome; you can email me at `C.vanReeuwijk@its.tudelft.nl`.

There is a Spar website at `www.pds.twi.tudelft.nl/timber/spar/overview.html`, and a website of the Timber compiler at `www.pds.its.tudelft.nl/timber` There is also a public mailing list with announcements of Spar and the Timber compiler. To subscribe, send an email to `spar-request@pds.its.tudelft.nl` with the subject '`subscribe`'. To unsubscribe, send an email to `spar-request@pds.its.tudelft.nl` with the subject '`unsubscribe`'.

Chapter 1

Introduction

Spar is a programming language for high-performance computing, including parallel programming. Since high-performance computing often means computations on arrays, there is special support for arrays, including a 'toolkit' to build support for more specialized array types.

As stated, Spar supports parallel programming. The first question to answer is: why not leave the whole problem of parallel programming to the compiler? After all, normal compilers do a good job of mapping high-level code to machine instructions, so why can't the compiler map to multiple streams of machine instructions? There are a number of reasons for that:

1. Algorithms often contain so many data dependencies that they do not lend themselves to automatic parallelization. Automatic and unattended transformation to an algorithm with less dependencies is far beyond the current state of the art in compiler technology. Therefore, the programmer often has to choose a different algorithm or coding to exploit parallelism.

   For example: consider a linear search through an array for an arbitrary element with the given value. On a parallel computer system we could speed up the search by letting each processor search a part of the array, but this will not always yield the same result as the linear search: it may not return the *first* element with that value. Depending on the application, we may, or may not, care about the difference.

   In other words, the sequential program may be *over-specified*.

2. Automatic load balancing is difficult, because (a) predicting execution time exactly is impossible (it is equivalent to solving the halting problem)[1], and (b) placing the tasks optimally on the available processors is NP-hard.

   For example: the linear search mentioned above may be known to usually succeed in the first few entries, or may be known to fail often. In general, this is impossible to know beforehand.

3. Communication (either implicit or explicit) makes the problem even more difficult, because it introduces new freedom, and requires the prediction of the performance of the communication system.

[1] However, it is often possible to give an *approximation* of the execution time that is accurate enough for practical purposes.

Despite these obstacles, there have been some efforts to do automatic parallelization, but these efforts necessarily only solve an approximate problem, make drastic assumptions, or restrict themselves to smaller domains. We must therefore accept that writing a program for a parallel computer requires the programmer to express parallelism explicitly.

Although the need has been there for a long time, there is still no programming language that allows parallel programs to be expressed conveniently. This means that writing a parallel program still requires a lot more effort than writing a sequential program. Obviously, this has not helped the acceptance of parallel computers.

Apart from the problems of sequential programs, a programmer of a parallel program has to cope with new hazards:

- Deadlock.
- Non-determinism (e.g. race conditions).
- Parallelization overhead.
- Resource utilization.

As we have seen, compilers cannot cope with these problems automatically. The programmer will have to tackle the same problems, which are hard for him/her too, but apparently we humans are able to cope. Nevertheless, a good programming language should provide as much support as possible to minimize the impact of these hazards. Many existing programming languages that provide explicit parallelism fail in this respect, because they provide only very primitive constructs. For example, Occam[13], Java[6], and CC++[3] all provide constructs that make it very easy for the programmer to cause deadlock or race conditions. Languages such as HPF[10] are not dangerous in this respect, but, because of their limited parallel programming model, make it more difficult to avoid parallelization overhead and to maximize resource utilization.

In Spar it is tried to avoid both these shortcomings by confining the programmer to the class of parallel programs known as SPC (sequential/parallel with contention) programs. It has been conjectured [5] that any parallel program can be rewritten to a SPC program, with an overhead of at most a factor two, and usually much lower. Obviously, this conjecture is difficult to prove in the general case, but compelling evidence has been gained that suggests the loss of parallelism caused by this restriction is limited [4].

A nice property of SPC programs is that fairly accurate performance predictions can be made, which makes it easier for the compiler to maximize resource utilization [5]. Another nice property is that they are inherently free of deadlock. Moreover, the Spar language constructs have been designed to limit non-determinism.

The Spar/Java programming language is designed as a modern programming language for parallel computer systems. It provides all the conveniences of modern programming languages, plus good support for parallel programming, while remaining sufficiently close to the implementation level to allow compilation to efficient code. Spar/Java is almost a superset of Java[6], although there are some aspects of Java that are not supported. See section 1.1 for a more detailed overview of the differences.

At the places where Spar extends Java, the language constructs were often inspired by functional or other non-imperative languages. In particular, tuples are copied from functional languages such as ML [9], Miranda [18, 19] and Haskell [11]. The concept of tuple indexing and the embedding of macros in the language, and the concept of types as parameters is from the more fundamental language theory of Raymond Boute [1] and from the FORFUN project [16]. Spar is not the first language to add parameterized types to Java: In their language Pizza [14], Martin Odersky and Philip Wadler add parametric polymorphism, higher-order functions and algebraic types to Java.

Since a parallel language is often used to express numerical algorithms and other algorithms that work on arrays, a parallel language should provide substantial support for arrays. Spar provides this. It is largely inspired by Fortran 90 [12] and Booster [15].

1.1 Current status

This document describes the Spar language extensions to Java. Appendix B also lists the restrictions of version 1.5 of the Timber compiler, which implements Spar/Java. For information purposes, appendix C describes the planned future extensions of Spar.

We use the second edition of the Java Language Specification (or JLS2 for short) [8], as the basis of our extensions.

Chapter 2

Lexical structure

2.0.1 keywords

Next to all the Java keywords listed in JLS2 3.9, Spar also reserves the following keywords[1]. Therefore, they cannot be used as identifiers.

Keyword: one of

```
__delete __print __println __string complex each foreach
globalpragmas inline pragma type
```

It is possible to instruct the Spar compiler with a command-line option to ignore some of the Spar extensions. This reverts the relevant keywords to normal symbols. See §A.2.3.

The new keyword `assert`, introduced in Java 1.4, is also reserved as a keyword, but not yet supported.

2.0.2 Literals

In addition to the Java literals described in JLS2 3.10, Spar supports complex literals. These have the same syntax as *FloatingPointLiteral* (see JLS2 3.10.2), except that they have the suffix `i` or `I`.

For example, `1i` is a valid imaginary literal.

This extension introduces a small incompatibility, because a valid expression such as `""+1instanceof String` will be parsed differently from Java. Simply rewriting the expression to `""+1 instanceof` is enough to fix this.

[1] Future versions of Spar will likely also use the keyword `wait`.

Chapter 3

Types, values and variables

3.1 Kinds of Types and Values

Next to primitive types and reference types, Spar adds a new kind of type, tuples. Thus, the definition of *Type* becomes (compare this with JLS2 4.1):

> *Type*:
> *PrimitiveType*
> *TupleType*
> *ReferenceType*

Tuple types are described in Chapter 5.

3.2 Complex numbers

As an extension to Java, Spar supports the primitive type `complex`. There are no `complex` literals, only imaginary literals (written, for example, as `2.0i`).

A `complex` value can also be constructed with the following expression:

> *PrimaryNoNewArray*:
> `complex (` *Real-Expression* `,` *Imag-Expression* `)`

Internally, complex numbers are represented as a pair of `double` numbers for the real and imaginary part.

3.2.1 Complex Operations

Spar provides a number of operators that act on complex values:

- The numerical equality operators `==` and `!=`; they result in a value of type `boolean`.
- The numerical operators, which result in a value of type `complex`:
  - The unary plus and minus operators `+` and `-`.
  - The multiplicative operators[1] `*` and `/`.

[1] The operator "`%`" is meaningless on complex numbers, so is *not* supported.

- The additive operators + and -.
  - The increment operator ++, both prefix and postfix.
  - The decrement operator --, both prefix and postfix.
- The conditional operator ?:.
- The cast operator, which can convert from a `complex` value to a value of any specified numeric type. It always uses the real part of the value.
- The string concatenation operator +, which, when given a `String` operand and a `complex` operand, will convert the `complex` operand to a `String` representing its value in decimal form, and then produces a newly created `String` by concatenating the two strings.

For example, the program:

```
public class cplx {
    public static void main( String args[] ){
        complex c = complex( 2, 1 );
        System.out.println( "c="+c );
        c++;
        System.out.println( "c="+c );
        c *= 2;
        System.out.println( "c="+c );
        c -= complex( 1, 2 );
        System.out.println( "c="+c );
        c *= 2+3i;
        System.out.println( "c="+c );
    }
}
```

generates the following output:

```
c=(2.0,1.0)
c=(3.0,1.0)
c=(6.0,2.0)
c=(5.0,0.0)
c=(10.0,15.0)
```

3.3 Variables

A variable of a tuple type always holds a value of that exact tuple type.

The initial value of a tuple is a tuple with the initial values of the elements of the tuple. For example, the program:

```
class Point {
    [int,int] coord;
}

public class initialtuple {
    public static void main(){
```

```
        Point p = new Point();

        System.out.println( "p.coord[0] = " + p.coord[0] );
    }
}
```

prints:

```
p.coord[0] = 0
```

illustrating the default initialization of `coord`, which occurs when a new instance of type `Point` is constructed.

Chapter 4

Conversions and promotions

4.1 Kinds of Conversion

4.1.1 Widening Primitive Conversions

Next to the widening primitive conversions described in JLS2 5.1.2, Spar supports the following:

- `byte`, `short`, `char`, `int`, `long`, `float` or `double` to `complex`

For example, the following program contains a widening conversion from `int` to `complex`:

```
public class widecplx {
    public static void main( String args[] ){
        int n = 1;
        complex c = n;  // Widening conversion to complex
        System.out.println( "c="+c );
    }
}
```

It generates the following output:

```
c=(1.0,0.0)
```

4.1.2 Narrowing Primitive Conversions

Next to the narrowing primitive conversions described in JLS2 5.1.3, Spar supports the following:

- `complex` to `byte`, `short`, `char`, `int`, `long`, `float` or `double`

For example, the following program contains a narrowing conversion from `complex` to `int`:

```
public class castcplx {
    public static void main( String args[] ){
        complex c = 3+2i;
        int n = (int) c;   // Narrowing conversion.
        System.out.println( "n="+n );
    }
}
```

It generates the following output:

```
n=3
```

4.1.3 Tuple conversions

A tuple conversion converts each element of a tuple to its target type. Tuple conversions are only allowed between tuples of the same length.

If there is at least one element of the tuple that requires narrowing conversion, the conversion is called a *narrowing tuple conversion*. Otherwise, if there is at least one element of the tuple that requires widening conversion, the conversion is called *widening tuple conversion*. Otherwise, the conversion must be an identity conversion.

For example, the following program contains a widening tuple conversion:

```
public class castcplx {
    public static void main( String args[] ){
        complex c = 3+2i;
        int n = (int) c;   // Narrowing conversion.
        System.out.println( "n="+n );
    }
}
```

It generates the following output:

```
n=3
```

4.2 Numeric Promotions

In addition to the binary numeric promotions described in JLS2 5.6.2, Spar supports the following binary numeric promotion:

- If either operand is of type `complex` the other is converted to `complex`.

For example, the following program contains a promotion from `int` to `complex`:

```
public class promocplx {
    public static void main( String args[] ){
        complex c1 = 3+2i;
        int n = 12;
        complex c2 = c1+n;  // 'n' is promoted to complex
        System.out.println( "c2="+c2 );
    }
}
```

The program generates the following output:

```
c2=(15.0,2.0)
```

4.3 Binary tuple promotion

If one of the operands of a binary operator is a tuple, and the other one is a scalar, the scalar operand is promoted to a tuple through *element replication*: the scalar expression is evaluated once, and the result is used repeatedly to fill the fields of a tuple with the same length as the other operand.

For example, the following program contains a binary tuple promotion from from `int` to `[int,int]`:

```
public class promotuple {
    public static void main( String args[] ){
        [int,int] a = [2,3];
        final int i = 2;

        // 'i' is subject to binary tuple promotion
        [int,int] b = i*a;
        System.out.println( "b[0]="+b[0]+", b[1]="+b[1] );
    }
}
```

The program generates the following output:

```
b[0]=4, b[1]=6
```

Chapter 5

Tuples

A tuple is a list of elements. The list is of fixed size, and each element can be of any type. Tuples can be constructed by surrounding a list of expressions with square brackets. For example, `[1,'a']` constructs a tuple of two elements. Such an expression is called an *explicit tuple*. Explicit tuples have the following syntax:

Expression:
  `[` $Expression_{list}$ `]`

The type of a tuple has the following syntax:

type:
  `[` $VerboseType_{list}$ `]`

VerboseType:
  PrimitiveType $Pragmas_{opt}$
  TupleType $Pragmas_{opt}$
  `type` *Type*

The types in a tuple specification must be preceded with the keyword `type` to distinguish them from variable names. In cases where no ambiguity is possible (primitive types and tuples), this keyword can be left out. For example, the following is a valid declaration and initialization of a variable of a tuple type:

```
[type int, type int, type Object] a = [1,1,null];
```

Since for primitive types the `type` keyword can be omitted, the following is equivalent:

```
[int, int, type Object] a = [1,1,null];
```

Since a tuple usually contains elements of primitive types, this allows for a compact notation.

Tuples have a field `length` that represents the length (the number of elements) of the tuple. This expression is a compile-time constant.

The following program demonstrates the use of tuples:

```
public class tupledemo {
    public static void main(){
        [int,double,char] x = [0,0.0,'\0'];

        x = [1,2.0,'x'];
        System.out.println( x[0] + " " + x[1] +
                               " " + x[2] + " " + x.length );
    }
}
```

This will produce the following output:

```
1 2.0 x 3
```

Just like primitive types, tuples do not have to be created explicitly (i.e. you do not have to do '`new`' for them), and just like primitive types, they are passed by value. A tuple of length 1 is *not* the same as a scalar.

5.1 Vector tuples

A *vector tuple* is a tuple where all elements are of the same type. For the type of such a vector tuple there is a special notation:

type:
  [*VerboseType* ^ *expression*]

The following program demonstrates the use of vector tuples:

```
public class vectordemo {
    public static void main(){
        [int^3] x = [0,0,0];

        System.out.println( x[0] + " " + x[1] + " " + x[2] );
        x[0] = 1;
        x[1] = 2;
        x[2] = 3;
        System.out.println( x[0] + " " + x[1] + " " + x[2] );
        x = [2,3,1];
        System.out.println( x[0] + " " + x[1] + " " + x[2] );
    }
}
```

This will produce the following output:

```
0 0 0
1 2 3
2 3 1
```

As will be shown later in Chapter 8, vector tuples are important for array access, and for the Array interface.

5.2 Tuple matching expressions

LeftHandSide:
  [$LeftHandSide_{list}$]

An explicit tuple can be used at the left-hand side of an assignment. The tuple should only contain expressions that may occur at the left-hand side themselves. For example, the following program demonstrates the use of tuple matching:

```
public class tuplematch {
    public static void main(){
        [int,double,char] x = [0,0.0,'\0'];
        int a;
        double b;
        char c;

        x = [1,2.0,'x'];
        [a,b,c] = x;

        System.out.println( a + " " + b + " " + c );
    }
}
```

This will produce the following output:

```
1 2.0 x
```

Chapter 6

Classes

6.1 Parameterized classes

Spar generalizes Java classes by allowing class definitions to be parameterized.
For example, consider this class, taken from [2]:

```
class Stack extends Object {
    static final int STACK_EMPTY = -1;
    Object[*] stackelements;
    int topelement = STACK_EMPTY;

    void push( Object e ){
        stackelements[++topelement] = e;
    }

    Object pop() {
        return stackelements[topelement--];
    }

    boolean isEmpty(){
        return ( topelement == STACK_EMPTY );
    }
}
```

The stack class above can hold elements of arbitrary type. This might be useful, but in some circumstances, for example for stricter type checking or for more efficiency, we want to restrict the stack to elements of a given type. We could implement lots of classes, such as IntStack, StringStack and PointStack, but it is much more useful to implement a generic stack that gets the type of elements it must stack as parameter. In Spar this is possible as follows:

```
class TypedStack(| type t |) {
    static final int STACK_EMPTY = -1;
    t[*] stackelements = new t[100];
    int topelement = STACK_EMPTY;

    public TypedStack() {}
```

```
    public void push( t e ){
        stackelements[++topelement] = e;
    }

    public t pop() {
        if( topelement == STACK_EMPTY )
            return (t) 0;
        else {
            return stackelements[topelement--];
        }
    }

    public boolean isEmpty(){
        if( topelement == STACK_EMPTY )
            return true;
        else
            return false;
    }
}

public class typedstack {
    public static void main(){
        TypedStack(| char |) s = new TypedStack(| char |)();

        s.push( 'a' );
        s.push( 'b' );
        char c = s.pop();
        System.out.println( c );
    }
}
```

This program will produce the following output:

```
b
```

This program defines a parameterized class `TypedStack` with a parameter `t` representing the type of elements to be stacked. It then defines an instance `s` of a `TypedStack` for `char` elements, it pushes two elements on the stack, pops one from the stack, and prints it.

To accommodate parameterized classes, the syntax of JLS2 8.1 is generalized to:

> *ClassDeclaration*:
> $\quad$ $\mathit{ClassModifiers}_{opt}$ `class` *Identifier* $\mathit{TypeParameters}_{opt}$ Super_{opt} $\mathit{Interfaces}_{opt}$ *ClassBody*

> *TypeParameters*:
> $\quad$ `(|` $\mathit{FormalParameter}_{list}$ `|)`

The Java grammar rule for `ClassOrInterfaceType` is generalized to:

ClassOrInterfaceType:
    Name
    GenericClassOrInterfaceType

GenericClassOrInterfaceType:
    Name `(|` $Argument_{list}$ `|)`

and *Argument* is generalized to also allow types as actual parameter:

Argument:
    Expression
    VerboseType

The grammar of formal parameters is extended with an additional rule:

FormalParameter:
    $Modifiers_{opt}$ `type` *Identifier.*

The actual parameters of a class may be types (the corresponding formal parameter must have type `type`), or values of a primitive type[1]. Actual parameters must evaluate to compile-time constants.

For each different combination of actual parameters, a new instance of the parameterized class is constructed. These class instances are then treated as classes in the same package as the original, parameterized, class.

Type parameters of parameterized classes are hidden by nested formal type parameters. Value parameters are hidden by nested formal parameters, except formal type parameters. They are also hidden by nested declarations, including cardinality variables.

As the extension to the grammar of *FormalParameter* implies, methods and constructors may also have `type` formal parameters. These are only allowed if the method or constructor is declared `inline`, see §6.2.

6.2 Inlining

Spar provides inlining for two different reasons: (a) to allow the abstraction of simple constructs without paying the cost of a function call, and (b) to allow type abstraction.

An inlined method or constructor is similar to an ordinary method or constructor, but it is declared to be a inlined with the `inline` keyword. For example:

```
class Stats {
    long sum;
    int n;

    inline Stats() { sum = 0; n = 0; }

    inline void update( int val ) { n++; sum += val; }
```

[1] Class parameters cannot be of reference types, since the equality of two reference cannot be determined at compile-time. For similar reasons, floating point class parameters are dubious, although they are allowed.

```
    inline float average() {
        return ((float) val)/((float) n);
    }
}
```

The methods and constructors in this class are very similar to ordinary methods, but the compiler is required to expand the `inline` methods compile time[2]. This usage of the construct is very similar to the `inline` construct of C++, except that here the user gives an *order* instead of a *hint*.

An inlined method must be either `static` or `final`, inlined methods cannot be `native` or `abstract`.

[2]The compiler has the liberty to expand other methods as well, but this is up to the compiler.

Chapter 7

Interfaces

7.1 Parameterized interfaces

Spar generalizes Java interfaces by allowing the interface definition to be parameterized.

For example, consider the following interface from [2]:

```
interface Collection {
    int MAXIMUM = 500;

    void add( Object obj );
    void delete( Object obj );
    Object find( Object obj );
    int currentCount();
}
```

A class can now promise to implement all methods of this interface by declaring:

```
Class Bag implements Collection {
    . . .
};
```

In Spar, interfaces can be parameterized:

```
interface TypedCollection(| type t |){
    void add( t obj );
    void delete( t obj );
    t find( t obj );
    int currentCount();
}

class TypedBag(| type t |) implements TypedCollection(| type t |) {
    t[*] elements = new t[0];
    int sz = 0;

    public TypedBag() {}
    private int search( t elm ){
        for( i :- 0:sz ){
```

```
            if( elm == elements[i] ){
                return i;
            }
        }
        return -1;
    }
    public void add( t e ){
        if( elements.length<=sz ){
            t[*] newelements = new t[2*elements.length+1];
            for( i:- 0:sz ){
                newelements[i] = elements[i];
            }
            elements = newelements;
        }
        elements[sz++] = e;
    }

    public void delete( t e ) {
        int pos = search( e );
        if( pos != -1 ){
            for( i :- pos+1:sz-1 ){
                elements[i] = elements[i+1];
            }
            sz--;
        }
    }

    public int currentcount(){ return sz; }
}

public class typedinf {
    public static void main(){
        TypedBag(| type char |) s = new TypedBag(| type char |)();

        s.add( 'a' );
        s.add( 'b' );
        s.add( 'b' );
        System.out.println( s.currentcount() );
        s.delete( 'b' );
        System.out.println( s.currentcount() );
    }
}
```

This will produce the following output:

```
3
2
```

To accommodate parameterized types, the grammar of *InterfaceDeclaration* is generalized to:

InterfaceDeclaration:

InterfaceModifiers$_{opt}$ `interface` *Identifier* *TypeParameters*$_{opt}$ *ExtendsInterfaces*$_{opt}$ *InterfaceBody*

See §6.1 for the definition of *TypeParameters* and related extensions of the grammar.

An important application of typed parameterized interfaces is the Array interface, see §8.6.

Chapter 8

Arrays

Since Spar/Java is a language for high-performance computation, it has far more extensive support for arrays than Java. To remain compatible with Java, Spar recognizes and embeds Java arrays.

Spar generalizes Java arrays on the following points:

- Arrays can be multi-dimensional.
- Arrays can be distributed over multiple processors.
- Arrays can be subscripted with `int` vector expressions.

Also, a number of other language extensions were designed to contribute to a 'toolkit' of extensions for the construction of specialized array types.

8.1 Array types

An array type is written as the name of an element type, followed by a number of abstract shape specifications. For example:

```
int[*] v;                // A 1-dimensional array
int[*,*] A;              // A 2-dimensional array
int[] n;                 // For Java compat., a 1-dimensional array
```

Alternatively, these array declarations may be written as:

```
int v[*];                // A 1-dimensional array
int A[*,*];              // A 2-dimensional array
int n[];                 // For Java compat., a 1-dimensional array
```

These two styles of declaration are completely equivalent.

The number of dimensions of an array, called the *rank* of the array, is specified by the number of `*` in the list. To remain compatible with Java, an empty list is not interpreted as a zero-dimensional array, but as a one-dimensional array. Thus, the declarations `int a[]` and `int a[*]` are equivalent.

Spar also allows an alternative notation to specify the rank. For example, the two type expressions `int [*,*]` and `int [*^2]` are equivalent. In general an arbitrary expression is allowed after the '`^`', provided that it is a non-negative compile-time constant of type `int`. This notation is much more flexible; for example, it allows the rank of an array to be dependent on a parameter of a parameterized type.

8.2 Array creation

A variable of an array type holds a reference to an object. Declaring a variable of an array type does not create an array object or allocate any space for array components. It creates only the variable itself, which can contain a reference to an array.

However, the initializer part of a declarator may create an array, a reference to which then becomes the initial value of the variable.

Because an array's length is not part of its type, a single variable of an array type may contain references to arrays of different lengths.

In an array creation expression, the list of sizes of the array is a vector tuple. Next to the standard, immediate, specification of the vector, Spar also allows specification with an arbitrary vector tuple expression, using the `@` operator, see the example below.

Here are some examples of declarations of array variables that create array objects:

```
public class arrcreate {
    public static void main(){
        int a[*] = new int[4];
        short b[*,*] = new short[6,8];
        int c[*] = new int[] { 1, 2, 3, 4 };
        int sq[*] = { 1, 4, 9, 16, 25, 36 };
        float ident[*,*] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
        float vv[*][*] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
        String[] aos = { "array", "of", "string" };
        [int^2] v = [12,12];
        int d[*,*] = new int@v;
    }
}
```

Note that `ident` and `vv` have the same initialization expression, but they are *not* equivalent. The first is a two-dimensional array, the second is a one-dimensional array of one-dimensional arrays.

8.3 Array access

A component of an array is accessed by an array access expression that consists of an expression whose value is an array reference followed by an explicit `int` vector, as in: `A[i,j]`. All arrays are 0-origin. A one-dimensional array with length n can be indexed by the integers 0 to $n-1$.

In an array subscript expression such a `A[1,2]`, the expression `[1,2]` is considered an explicit `int` vector tuple expression (see Chapter 5 for a discussion of tuples). Spar generalizes Java to allow subscription with arbitrary `int` vector tuple expressions.

The most obvious way to implement this generalization would be to interpret juxtaposition as subscript operator. Thus, given an array `A` and an `int` vector tuple `v`, the expression `A v` would represent an array access. Unfortunately, such an expression is ambiguous in the context of many Java expressions. Therefore, an explicit subscript operator `@` is introduced. Thus, `A@v` is a valid array access. Since an explicit vector is also a vector expression, an array access such as `A@[1,2]` is also valid.

The `@` operator has the same high precedence as unary operators, so that expressions such as `a@v+1` is evaluated as `(a@v)+1`.

A tuple can be used for indexing if its length is equal to the rank of the indexed array, and if all elements of the tuple can be converted to type `int` through unary numeric promotion.

For example, the following assigns 2 to array element `[2,3]` of array `A`, and assigns 5 to array element `[3,2]`.

```
public class arrassign {
    public static void main(){
        int[*,*] A = new int[4,4];      // A 2-dim array.
        A[2,3] = 2;
        A@[3,2] = 5;
        for( int x=0; x<A.getSize(0); x++ ){
            for( int y=0; y<A.getSize(1); y++ ){
                System.out.print( A[x,y] + " " );
            }
            System.out.println();
        }
    }
}
```

This program will produce the following output:

```
0 0 0 0
0 0 0 0
0 0 0 2
0 0 5 0
```

8.4 The `length` field

Java arrays have a `length` field, that contains the length of the array. For multi-dimensional arrays this is the product of all dimension lengths.

Use the method `getSize(int)` to get the size of a dimension of a multi-dimensional array, or the method `getSize()` to get a vector tuple with all sizes.

8.5 The `getSize` method and `getRoom` methods

In Spar/Java every array supports the following methods:

```
int getSize( int n );
[int^rank] getSize();
int getRoom();
```

where `rank` is the rank of the array. The first method returns the size of the array in the given dimension, the second method returns a vector with all sizes.

For example, the following program creates a two-dimensional array, and fills it with zeroes:

```
public class arrzero {
    public static void main(){
        int[*,*] A = new int[10,10];

        for( int x=0; x<A.getSize(0); x++ )
            for( int y=0; y<A.getSize(1); y++ )
                A[x,y] = 0;
    }
}
```

The following program is equivalent to the previous example, but uses vector notation:

```
public class arrzero1 {
    public static void main(){
        int[*,*] A = new int[10,10];

        for( v :- [0,0]:A.getSize() )
            A@v = 0;
    }
}
```

8.6 Overloading the subscript operator

The subscript operator (both the explicit `@` operator and the implicit subscript operator) can be used on class elements. In the context of an assignment, the assignment and subscript expression are translated to an invocation statement of the method `storeElement`, otherwise a subscript expression is translated to an invocation expression of the method `getElement`.

For example, the statement

```
a@v = x;
```

is translated to:

```
a.storeElement( v, x );
```

And the statement

```
x = a@v;
```

is translated to:

```
x = a.getElement( v );
```

It is recommended that classes that want to use this feature implement the interface Array. See §13.3 for details. Similarly, it is recommended that classes that implement arrays that can be grown and shrunk implement the interface ElasticArray.

For example, the following class defines a 'view' on the diagonal of a two-dimensional array.

```
class DiagonalView(| type t |) implements Array(| t, 2 |)
{
    t[*,*] ref;         // Reference to the viewed array

    inline DiagonalView( t[*,*] a ){ ref = a; }

    inline t getElement( [int] ix ){ return ref[ix[0],ix[0]]; }

    inline void storeElement( [int] ix, t elm )
    {
        ref[ix[0],ix[0]] = elm;
    }

    inline [int] getSize() {
        [int^2] dims = ref.getSize();
        // The brackets construct a vector, as required by the interface.
        return [Math.min( dims[0], dims[1] )];
    }
}
```

This class can now be used as follows:

```
int[*,*] a = new int[5,5];
DiagonalView(int) v = DiagonalView( a );
for( i=[0,0]:a.getSize() ){
    a@i = i[0]+i[1];
}
for( i :- 0:v.getSize(0) )
    v[i] = 0;
```

This will construct a matrix ‘a’, with each element set to the sum of its coordinates. The last statement then fills the elements of the diagonal with 0.

As another example, the following class implements a transpose view on an array of arbitrary size.

```
class TransposeView(| type t, int n |) implements Array(| t, n |)
{
    t [*^n] ref;          // Reference to the viewed array

    inline TransposeView( t [*^n] a ){ ref = a; }

    inline t getElement( [int^n] ix )
    {
        return ref@revVector( n, ix );
    }

    inline void storeElement( [int^n] ix, t elm )
    {
        ref@revVector( n, ix ) = elm;
    }
```

```
    inline [int^n] getSize()
    {
        return revVector( n, ref.getSize() );
    }

    static inline [int^n] revVector( int n, [int^n] v )
    {
        int ix;
        [int^n] res;

        for( ix=0:n ){
            res[(n-ix)-1] = v[ix];
        }
        return res;
    }
}
```

Note that this class even works for 0-dimensional and 1-dimensional arrays.

Chapter 9

Execution

9.1 Finding the class that contains the main method

According to the Java Language Specification, executing a Java program consists of executing the method with signature

```
public static void main( String args[] )
```

in a `.class` file that is given as parameter upon startup. See JLS2 12.1 for further details. In a static compiler this rule cannot be implemented, since `.class` files are never generated, and the compiler only gets the name of the `.spar` or `.java` file to compile. This makes a difference for Java source files such as

```
class foo {
    public static void main( String[] args ) {
        System.out.println( "Hi from foo" );
    }
}

public class bar {
    public static void main( String[] args ) {
        System.out.println( "Hi from bar" );
    }
}
```

Since there is a public class `bar` in this source file, this code must live in a source file with the name `bar.spar` or `bar.java`.

If you use the normal `Java` compiler on this file, two files will be generated: `foo.class` and `bar.class`. You can invoke either of the `main` methods by running one of the two `.class` files.

In a static compiler this is not possible, since you only specify the `.spar` or `.java` file to compile. Therefore, we specify that a compiler should search for the first public class in this file, and execute the `main` method in that class. Therefore, the `main` method in class `foo` cannot be used as the starting method.

If there is a compelling reason to do so, this could be refined by allowing the user to specify the main class (e.g. as a compiler option).

9.2 Alternative `main` method

As described above, a Spar compiler should search for the initial method to invoke in the first public class of the top-level source file it is given. In this class it searches for a method with the signature:

```
public static void main( String args[] )
```

Contrary to standard Java compilers, if such a method cannot be found, a Spar compiler should search the class again for a method with the signature:

```
public static void main()
```

For example, the following program is valid in Spar, but not in Java:

```
public class emptymain {
    public static void main(){
        System.out.println( "Hello world" );
    }
}
```

It will produce the following output:

```
Hello world
```

Chapter 10

Blocks and Statements

10.1 The `for` statement

Next to the `for` statement described in JLS2 14.13, Spar allows `for` loops with *cardinality lists*; the same notation that is used for the `foreach` statement. Thus, code like this:

```
public class forcard {
    public static void main(){
        int sum = 0;

        for( i :- 0:10 )
            sum += i;
        System.out.println( sum );
    }
}
```

is allowed. The program will print:

```
45
```

This form of the `for` statement is allowed for reasons of symmetry with the `foreach` statement. Moreover, this form allows easier loop analysis, because the bounds and stride of the loop are only evaluated once, and assignment to the loop variable is not allowed.

See §10.4 for further details on the syntax and meaning of cardinality lists.

Spar allows `for` loops to be unrolled explicitly, by annotating the loop with the `inline` modifier. Such an `inline for` statement must have a cardinality list with only compile-time constants.

The `inline for` statement is necessary for operations on tuples, and is also useful to force loop unrolling for performance reasons.

For example, the following loop:

```
inline for( i :- 0:4 ) a[i] = i;
```

is expanded by the compiler to:

```
a[0] = 0;
```

```
a[1] = 1;
a[2] = 2;
a[3] = 3;
```

10.2 Parallel programming

Spar/Java is intended for parallel programming. This requires the identification of code fragments that can be executed in parallel. A Spar/Java compiler will not try to determine these itself, but expects the programmer to describe them explicitly using special language constructs.

To expose parallelism to the compiler, two new language constructs are provided, the `each` statement, and the `foreach` statement.

10.3 The `each` statement

The `each` statement is similar to the `par` statement of Compositional C++[3]. Given a block such as:

```
each { s1; s2; }
```

The statements `s1` and `s2` are executed in arbitrary order. Once a statement is started, it must be completed before the next statement can be started. Thus, the compiler will choose one of the execution orders `s1; s2;`, or `s2; s1;`, even if the statements are compound.

10.4 The `foreach` statement

The `foreach` statement is a parameterized version of the `each` statement of the previous section. For example:

```
public class foreachcard {
    public static void main(){
        int sum = 0;

        foreach(i :- 0:10 )
            sum += i;
        System.out.println( sum );
    }
}
```

The program will print:

```
45
```

Similar to the `each` statement, once an iteration is started, it must be completed before the next iteration can be started. Thus, iterations cannot influence each other during their execution.

To allow easier analysis, the `foreach` has a range syntax rather than the traditional `while`-like syntax of the `for` statement of C.

For reasons of orthogonality Spar allows the range syntax in the `for` statement. The compiler is not required to attempt any parallelization, even for obvious cases. Spar does *not* support the `while`-like syntax in the `foreach` statement, since the behavior of such a loop cannot be defined properly.

> *statement*:
> `for` *Cardinalities Statement*
> `foreach` *Cardinalities Statement*
>
> *Cardinalities*:
> `(` $Cardinality_{list}$ $,_{opt}$ `)`
>
> *Cardinality*:
> *Identifier* `:-` $lower\text{-}Expression_{opt}$ `:` *upper-Expression*
> *Identifier* `:-` $lower\text{-}Expression_{opt}$ `:` *upper-Expression* `:` *stride-Expression*
> `(` *Cardinality* `)`

A scalar cardinality specifies an iteration range, consisting of a lower bound and an upper bound, and the name of the variable that iterates of this range. If the lower bound is left unspecified, 0 is assumed. If the stride is left unspecified, 1 is assumed.

A scalar cardinality declares the iterator variable as a `final int` variable, with the body of the `foreach` as scope.

The type of the lower bound, upper bound, and stride expressions must be an integral type, or a compile-time error occurs. Each expression undergoes unary numeric promotion (see JLS2 5.6.1). The stride should be positive. A negative or zero stride may lead to a compile-time error, if the compiler can detect it.

For example, the three loops in the following program all have the same result (every array element a_i is filled with i). However, the execution order of the statements may be different in the `foreach`, compared to the `for` versions of the loop.

```
public class loops {
    public static void main(){
        int a[] = new int[12];

        foreach( i :- 0:a.length )
            a[i] = i;
        for( i :- 0:a.length )
            a[i] = i;
        for( int i=0; i<a.length; i++ )
            a[i] = i;
    }
}
```

A cardinality list can contain more than one cardinality. For example, the following program:

```
public class multicard {
    public static void main(){
        for( i:- 2:5, j:- 1:3 ){
            System.out.print( i + "~" + j + " " );
```

```
        }
        System.out.println();
    }
}
```

contains a `for` loop with two cardinalities in its cardinality list. This program produces the following output:

```
2~1 2~2 3~1 3~2 4~1 4~2
```

The range of a `foreach` can also be described as a vector. For example, a two-dimensional array `b` would be initialized completely with:

```
foreach( i:-[0,0]:b.getSize() ){
    (b@i).init();
}
```

Note that this is not easy to express in the traditional `for(;;)` syntax, since there is no ordering comparison defined on vectors.

If a lower bound or stride vector is specified, it must have the same length as the upper bound.

Chapter 11

Expressions

11.1 Array Creation Expressions

Since Spar allows multi-dimensional arrays, the syntax for array creation expressions has been extended somewhat. In particular, the syntax of *ArrayCreationExpression* has been generalized to:

ArrayCreationExpression:
  `new` *PrimitiveType Vectors* $Dims_{opt}$
  `new` *PrimitiveType Vectors ArrayInitializer*
  `new` *ClassOrInterfaceType Vectors* $Dims_{opt}$
  `new` *ClassOrInterfaceType Vectors ArrayInitializer*
  `new` *TupleType Vectors* $Dims_{opt}$
  `new` *TupleType Vectors ArrayInitializer*

Vectors:
  Vector
  Vectors Vector

Vector:
  `[` $Expression_{list}$ `]`:

For example, the following program creates and fills a 2-dimensional array:

```
public class create2d {
    public static void main(){
        int a[*,*] = new int[4,4];

        for( i :- 0:4, j :- 0:4 ){
            a[i,j] = i+j;
        }
    }
}
```

11.2 Array Access Expressions

Obviously, array access expressions are more general than specified in JLS2 15.13, since Spar supports multi-dimensional arrays.

In Spar array subscripting is considered an operation of a vector expression on an array. It is expressed with the array subscript operator `@`, with the syntax:

OperatorExpression:
  OperatorExpression `@` *OperatorExpression*

The left-hand side of the expression should be an array, the right-hand side of the expression should be a tuple of the correct length. Each element of the tuple should be of type `int`; `short`, `byte`, or `char` elements may also be used, because they are subjected to unary numeric promotion, and become `int` values. Elements of type `long` or other non-integral types are not allowed by the compiler.

For example, the program:

```
public class vecsubscr {
    public static void main(){
        int[*,*] A = new int[3,4];      // A 2-dim array.

        for( v :- [0,0]:A.getSize() ){
            A@v = 0;
        }
        [int^2] i = [1,1];
        A@i = 1;
        A@[2,3] = 2;
        for( x :- 0:A.getSize(0) ){
            for( y :- 0:A.getSize(1) ){
                System.out.print( A[x,y] + " " );
            }
            System.out.println();
        }
    }
}
```

generates the following output:

```
0 0 0 0
0 1 0 0
0 0 0 2
```

For explicit vector expressions it is allowed to omit the `@` operator, in which case the expression degenerates to the familiar subscript expression.

ArrayAccess:
  Name Vector
  PrimaryNoNewArray Vector

Vector:
  `[` $Expression_{list}$ `]`

11.3 Unary operators

All unary operators also work on tuples. The operator is applied to each element of the tuple.

For example, the program:

```
public class unarytuple {
    public static void main(){
        [int^3] a = [1,2,3];
        [int^3] b = -a;         // Unary operator on tuple 'a'
        System.out.println( b[0] + " " + b[1] + " " + b[2] );
    }
}
```

generates the following output:

```
-1 -2 -3
```

11.4 Binary operators

All binary operators except the `instanceof` operator also work on tuples. The operator is applied to each element of the tuple. If one of the operands is a scalar, it is promoted to a tuple through *tuple conversion*, see §4.1.3.

For example, the program:

```
public class binarytuple {
    public static void main(){
        [int^3] a = [1,2,3];
        [int^3] b = 2*a;
        System.out.println( b[0] + " " + b[1] + " " + b[2] );
        [int^3] c = a-2;
        System.out.println( c[0] + " " + c[1] + " " + c[2] );
        [int^3] d = a*c;
        System.out.println( d[0] + " " + d[1] + " " + d[2] );
    }
}
```

generates the following output:

```
2 4 6
-1 0 1
-1 0 3
```

11.5 Relational Operators

11.5.1 Type Comparison Operator `instanceof`

In contrast to Java, Spar never causes a compile-time error on an `instanceof` expression (see JLS2 15.20.2) that always evaluates to `false`. Instead, it simply replaces the expression with `false`. Moreover, the type that is compared with can be an arbitrary type. For primitive types it is guaranteed that the expression is evaluated to a compile-time constant.

As a further extension, the `instanceof` can also be used to compare two types. Such an expression is always evaluated to a compile-time constant.

These extensions result in the following grammar for an `instanceof` expression:

InstanceOfExpression:
  RelationalExpression `instanceof` *Type*
  VerboseType `instanceof` *Type*

See §5 for the grammar of *VerboseType*.

The purpose of these extensions is to allow conditional compilation of code in parameterized classes. In particular, the expression `t instanceof Object` tests wether `t` is a reference type.

Chapter 12

Pragmas

Pragmas are intended to annotate program elements with information that is helpful to the compiler. A pragma should never influence the meaning of a program[1]; it should only reduce the compilation or execution time of the program.

Pragmas have the following syntax:

Pragmas:
  `<$` *Pragma*$_{list}$ `$>`

Pragma:
  Identifier
  Identifier `=` *PragmaExpression*

PragmaExpression:
  IntLiteral
  FloatLiteral
  DoubleLiteral
  StringLiteral
  `true`
  `false`
  Identifier
  `@` *Identifier*
  `(` *PragmaExpressions* `)`

PragmaExpressions:
  (empty)
  PragmaExpressions PragmaExpression

The following program demonstrates the use of several types of pragma. It uses the `Spar` language construct `globalpragma`, which is described below. The pragmas that are shown here suggest a certain interpretation, but they are only for illustration; it should not be assumed that these pragmas are recognized by any compiler engine.

[1]But note that the actual results of the program may differ if nondeterministic constructs (`each` or `foreach`) are used. Although the actual behavior of the program may have changed, its meaning has not.

```
globalpragmas <$
    entrypoint,         // A 'flag' pragma, without an expression
    processor=pentium2, // A pragma with a name as value
    boundscheck=false,  // A pragma with a boolean value
    optimizations=3,    // A pragma with a numerical value
    listingfile="pragmas.lst", // A pragma with a string value
    processors=(1 3 5), // A pragma with a list of numbers as value

    // A pragma with a more complicated structure as value
    mapping=(lambda (i j) (mod ((sum i j)) 3))
$>;

public class pragmas {
    public static void main( String[] args ) {
        int sum = 0;

        // A pragma that uses the '@' notation to refer to a variable
        // in the program (in this case variable 'i').
        foreach( i :- 0:23 ) <$ on=(mod @i 8) $> {
            sum += i;
        }
    }
}
```

12.1 Operators and subscripts in pragmas

As a convenience for the user, Spar allows the use of a number of binary operators. These binary operator expressions are internally translated to a normal pragma expression list. Spar also supports subscript-like expressions.

> *PragmaExpression*:
> *PragmaExpression PragmaBinop PragmaExpression*
> *PragmaExpression* `[` $PragmaExpression_{list}$ `]`
>
> *PragmaBinop: one of*
> `+ - * / % == != <= < >= >`

For example, the expression `a+b` is translated to `(sum a b)`. Operators are left-binding, which means that `a+b-c` is translated to `(subtract (sum a b) c)`. The expression `a[1,2]` is translated to `(at a 1 2)`.

Repeated use of the same operator, for example `a+b+c`, would result in nested lists such as `(sum (sum a b) c)`. For convenience sake such expressions are stratified into a single long list of operands. Thus, `(sum a b c)` is produced instead. This also applies if the nested expression is already in list format. Thus, `(sum a b)+c` is *also* translated to `(sum a b c)`.

Operators have the usual precedence, so that `a+b*i` is translated to `(sum a (prod b i))`, because the `*` operator as a higher precedence than the `+` operator.

You can use brackets to enforce operator precedence, but these brackets are not treated specially. For example, `(a+b)*i` is translated to `(prod ((sum a b)) i)`. Note the extra pair of brackets.

The supported operators, with their precedence and the expanded operator name, are listed below.

precedence	operator	name
0	`+`	`sum`
0	`-`	`subtract`
1	`*`	`prod`
1	`/`	`div`
1	`%`	`mod`
2	`==`	`eq`
2	`!=`	`ne`
2	`<=`	`le`
2	`<`	`lt`
2	`>=`	`ge`
2	`>`	`gt`
3	`[]`	`at`

For example:

```
globalpragmas <$
    entrypoint,         // A 'flag' pragma, without an expression
    processor=pentium2, // A pragma with a name as value
    boundscheck=false,  // A pragma with a boolean value
    optimizations=3,    // A pragma with a numerical value
    listingfile="pragmas.lst", // A pragma with a string value
    processors=(1 3 5), // A pragma with a list of numbers as value

    // A pragma with a more complicated structure as value
    mapping=(lambda (i j) (mod ((sum i j)) 3))
$>;

public class pragmas {
    public static void main( String[] args ) {
        int sum = 0;

        // A pragma that uses the '@' notation to refer to a variable
        // in the program (in this case variable 'i').
        foreach( i :- 0:23 ) <$ on=(mod @i 8) $> {
            sum += i;
        }
    }
}
```

12.2 Global pragmas

It is possible to annotate an entire program with a set of pragmas through the use of global pragmas:

GlobalPragmas:
  `globalpragmas` *Pragmas* ;

A global pragma declaration should be placed at the very start of a compilation unit, as shown in the following additional grammar rule for a compilation unit:

> *CompilationUnit*:
> *GlobalPragmas* *PackageDeclaration*$_{opt}$ *ImportDeclarations*$_{opt}$ *TypeDeclarations*$_{opt}$

For example:

```
globalpragmas <$ boundscheck=false $>;

public class globpragmas {
    public static void main( String[] args ) {
    }
}
```

12.3 Class and interface pragmas

Pragmas can be attached to class and interface declarations.

> *ClassDeclaration*:
> *Pragmas*$_{opt}$ *ClassModifiers*$_{opt}$ `class` *Identifier* *TypeParameters* *Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

> *InterfaceDeclaration*:
> *Pragmas*$_{opt}$ *InterfaceModifiers*$_{opt}$ `interface` *Identifier* *TypeParameters* *ExtendsInterfaces*$_{opt}$ *InterfaceBody*

For example:

```
<$ entrypoint $> public class classpragmas {
    public static void main( String[] args ) {
    }
}
```

Class and interface pragmas are only maintained in the frontend, and are not passed on to the parallelization engines. The frontend does not use these pragmas either; they are only provided for interpretation by a future version of the frontend.

12.4 Expression pragmas

Pragmas can be attached to expressions.

> *Expression*:
> *Pragmas* *Expression*

Pragma expressions have a very low precedence, so an expression such as `<$ volatile $> a+b` is interpreted as `<$ volatile $> (a+b)`. For example:

```
public class exprpragmas {
    public static void main( String[] args ) {
        int x = <$ constant $> 42;
    }
}
```

12.5 `new` expression pragmas

Pragmas can be attached to array creation expressions. They are mainly intended for distribution annotations for the arrays that are created.

Note that in contrast to most other pragmas, these follow *after* the expression they annotate. This is to allow them to be attached to each vector in the array creation expression.

ArrayCreationExpression:
`new` *Type Vectors* Dims_{opt}

Vectors:
Vector $\mathit{Pragmas}_{opt}$
Vectors Vector $\mathit{Pragmas}_{opt}$

For example:

```
public class newpragmas {
    public static void main( String[] args ) {
        int a[] = new int[42] <$ distribution="[block]" $>;
    }
}
```

12.6 Class and interface member pragmas

Pragmas can be attached to class and interface members.

ClassBodyDeclaration:
$\mathit{Pragmas}_{opt}$ *StaticInitializer*
$\mathit{Pragmas}_{opt}$ *ConstructorDeclaration*
$\mathit{Pragmas}_{opt}$ *Block*

FieldDeclaration:
$\mathit{Pragmas}_{opt}$ $\mathit{Modifiers}_{opt}$ *Type VariableDeclarators* ;

MethodDeclaration:
$\mathit{Pragmas}_{opt}$ $\mathit{MethodHeader}_{opt}$ *MethodBody*

AbstractMethodDeclaration:
$\mathit{Pragmas}_{opt}$ $\mathit{MethodHeader}_{opt}$;

For example:

```
public class memberpragmas {
    <$ replicated $> final static int n = 3;
    <$ entrypoint $> public static void main( String[] args ) {
    }
}
```

Pragmas on class instance variables are only maintained in the frontend, and are not passed on to the parallelization engines. The frontend does not use these pragmas either; they are only provided for interpretation by a future version of the frontend.

12.7 Type pragmas

Pragmas can be attached to types.

Note that in contrast to most other pragmas, these follow *after* the expression they annotate. This is to distinguish them from declaration pragmas.

Type:
  Type Pragmas

Dim:
  `[ ]` $\mathit{Pragmas}_{opt}$
  `[` *SizeList* $,_{opt}$ `]` $\mathit{Pragmas}_{opt}$

For example:

```
public class typepragmas {
    public static void main( String[] args ) {
        int <$ range=(0 20) $> x = 10;
    }
}
```

12.8 Declaration and statement pragmas

Pragmas can be attached to local declarations and statements.

Statement:
  $\mathit{Pragmas}_{opt}$ *LabelName* `:` *Statement*
  $\mathit{Pragmas}_{opt}$ *UnlabeledStatement*

LocalVariableDeclarationStatement:
  $\mathit{Pragmas}_{opt}$ *LocalVariableDeclaration* `;`

ExplicitConstructorInvocation:
  $\mathit{Pragmas}_{opt}$ `this (` $\mathit{ArgumentList}_{opt}$ `) ;`
  $\mathit{Pragmas}_{opt}$ `super (` $\mathit{ArgumentList}_{opt}$ `) ;`
  $\mathit{Pragmas}_{opt}$ *Primary* `. super (` $\mathit{ArgumentList}_{opt}$ `) ;`

For example:

```
public class statementpragmas {
    public static void main( String[] args ) {
        <$ dummy $> int x = 42;
        foreach( i :- 0:20 ) <$ on=@i%5 $> {
            x += i;
        }
    }
}
```

12.9 Formal parameter pragmas

Pragmas can be attached to formal parameter declaration.

FormalParameter:
  $Pragmas_{opt}$ $Modifiers_{opt}$ *Type* *VariableDeclaratorId*
  $Pragmas_{opt}$ $Modifiers_{opt}$ `type` *VariableDeclaratorId*

For example:

```
public class formalpragmas {
    public static void main( <$ replicated $> String[] args ) {
    }
}
```

Chapter 13

The standard library

13.1 The Class `java.lang.Complex`

The class `Complex` is the wrapper class for the `complex` primitive type, similar to standard Java classes such as `java.lang.Double`. It also provides some math functions similar to those in `java.lang.Math`.

```
public final class Complex extends Number {
    public Complex(complex value);
    public Complex(String s) throws NumberFormatException;
    public String toString();
    public boolean equals(Object obj);
    public int hashCode();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public complex complexValue();
    public static String toString(complex value);
    public static Complex valueOf(String s) throws NumberFormatException;
    native public static complex sin(complex a);
    native public static complex cos(complex a);
    native public static complex tan(complex a);
    native public static complex sqrt(complex a);
    native public static complex exp(complex a);
    native public static complex log(complex a);
    native public static double abs(complex a);
    native public static double arg(complex a);
    native public static double norm(complex a);
    native public static complex polar( double rho, double theta );
    native public static complex conj(complex a);
    native public static double real(complex a);
    native public static double imag(complex a);
    native public static complex pow(complex a, complex b);
}
```

13.1.1 public Complex(complex value)

This constructor initializes a newly created Complex object so that it represents the primitive value that is the argument.

13.1.2 public Complex(String s)

Interpret the given string as a complex number, and initialize a newly created Complex object with the value.

13.1.3 public String toString()

The primitive `complex` value represented by this Complex object is converted to a string exactly as if by the method `toString` on one argument (see §13.1.11).

Overrides the `toString` method of Object.

13.1.4 public boolean equals(Object obj)

The result is `true` if and only if the argument is not `null` and is a Complex object that represents the same `complex` value as this Complex object.

Overrides the `equals` method of Object.

13.1.5 public int hashCode()

For the moment, a rather weak hashing algorithm is used.

Overrides the `hashCode` method of Object.

13.1.6 public int intValue()

The real part of the `complex` value represented by this Complex object is converted to type `int` and the result of the conversion is returned.

Overrides the `intValue` method of Object.

13.1.7 public long longValue()

The real part of the `complex` value represented by this Complex object is converted to type `long` and the result of the conversion is returned.

Overrides the `longValue` method of Object.

13.1.8 public float floatValue()

The real part of the `complex` value represented by this Complex object is converted to type `float` and the result of the conversion is returned.

Overrides the `floatValue` method of Object.

13.1.9 public double doubleValue()

The real part of the `complex` value represented by this Complex object is returned.

Overrides the `doubleValue` method of Object.

13.1.10 public complex complexValue()

The complex value represented by this Complex object is returned.

13.1.11 public static String toString(complex value)

The argument is converted to a readable string format as if the expression

```
"("+Double.toString(real(v))+","+Double.toString(imag(v))+")"
```

is evaluated.

13.1.12 public static Complex valueOf(String s)

Given a string, interpret this as a complex number, and return the value.

13.1.13 public static complex sin(complex a)

This method computes an approximation to the sine of the argument.

13.1.14 public static complex cos(complex a)

This method computes an approximation to the cosine of the argument.

13.1.15 public static complex tan(complex a)

This method computes an approximation to the tan of the argument.

Note that this method is not implemented in some versions of the of the C++ `complex` libraries, and we rely that library.

13.1.16 public static complex sqrt(complex a)

This method computes an approximation to the square root of the argument.

13.1.17 public static complex exp(complex a)

This method computes an approximation to the exponential function of the argument.

13.1.18 public static complex log(complex a)

This method computes an approximation to the natural logarithm of the argument.

13.1.19 public static double abs(complex a)

For an argument $a = x + i \cdot y$, this method computes an approximation to $\sqrt{x^2 + y^2}$.

13.1.20 public static double arg(complex a)

This method computes an approximation to the angle of the argument.

13.1.21 public static double norm(complex a)

For an argument $a = x + i \cdot y$, this method computes $x^2 + y^2$. In other words, it computes the square of the value returned by the method `abs`.

13.1.22 public static complex polar(double rho, double theta)

Given a magnitude `rho` and an angle `theta` return the complex number constructed from these polar coordinates.

13.1.23 public static complex conj(complex a)

This method returns the conjugate of the argument. In other words, for an argument $a = x + i \cdot y$ it returns the value $a = x + i \cdot -y$.

13.1.24 public static double real(complex a)

This method returns the real part of the argument.

13.1.25 public static double imag(complex a)

This method returns the imaginary part of the argument.

13.1.26 public static complex pow(complex a, complex b)

This method computes an approximation to the mathematical operation of raising the first argument to the power of the second argument.

13.2 The Class `java.lang.String`

Next to the methods described for `java.lang.String`, Spar provides one other.

13.2.1 public static String valueOf(complex d)

A string is created and returned. The string is computed exactly as if by the method `Complex.toString` of one argument (see §13.1.11).

13.3 The Interface `spar.lang.Array`

Spar allows the subscript operators '`[]`' and '`@`' to work on any class, provided that the class implements the method `getElement` or `storeElement`, depending on the context. Although this is not enforced by Spar, it is recommended that such classes implement the interface `Array`, defined as follows:

```
interface Array(| type t, int rank |)
{
    t getElement( [int^rank] index ) throws IndexOutOfBoundsException;
    void storeElement( [int^rank] index, t elm )
        throws IndexOutOfBoundsException, ArrayStoreException;
```

```
    [int^rank] getSize();
}
```

Appendix A

Using the Timber compiler

This appendix describes how to use the Timber compiler, which implements Spar/Java.

A.1 Preparations

Timber is available for downloading from the Spar website. See www.pds.twi.tudelft.nl/timber/downloading.html for instructions on downloading and installation.

In this chapter we assume that the compiler has been installed.

A.2 Running the Spar compiler

It is traditional to check out a compiler by compiling and running a program that prints "Hello world". For Spar (and Java), the following program is sufficient:

```
public class hello {
    public static void main(){
        System.out.println( "Hello world" );
    }
}
```

To verify that the compiler is working as it should, create a file `hello.spar` with this contents[1], and run the compiler with:

```
spar hello.spar -o hello
```

This should produce a new executable `hello`. If you run it:

```
./hello
```

It should produce the familiar greeting.

[1] Remember that in Java it is required that a public class `hello` resides in a file `hello.java`. A Spar/Java compiler also allows it to reside in a file `hello.spar`.

A.2.1 Packages, compilation units and filenames

Each Spar compilation unit is stored as an individual source file. This source file must have a name ending in `.java` or in `.spar`. Moreover, each public class or interface `<object>` that is not an inner class or interface must reside in a file with the name `<object>.java` or `<object>.spar`.

If the compilation unit resides in an explicitly named package (that is, if it has a `package` statement), it must reside in a subdirectory that corresponds with the package name. The mapping between the package name and the subdirectory name is the same as the mapping is described in §7.2.1 of the Java Language Specification.

The subdirectory that corresponds with the package must reside in one of the directories in the *search path* of the Timber compiler. By default the following two directories form the search path:

- The directory `$prefix/lib/sparlib`[2], where `$prefix` is the directory that was specified as installation directory prefix during installation of the compiler.
- The directory `$KAFFEROOT/libraries/javalib`, where `$KAFFEROOT` represents the value of the environment variable of that name.

The directories are searched in the given order.

If the environment variable `SPARPATH` is set, is must contain a list of directories separated with '`:`'. These directories will also be searched, after the directories listed above.

For example, let us assume that the environment variable `KAFFEROOT` contains `/usr/local/kaffe-1.0.6`, the installation prefix was `/usr/local`, and the environment variable `SPARPATH` contains `/users/leo/sparpackages:/users/frits/sparpackages`. If the compiler now wants to compile a class `ZipDecoder` in the package `pds.utils`, it will try to open the following files in the given order:

- `/usr/local/lib/sparlib/pds/utils/ZipDecoder.spar`
- `/usr/local/lib/sparlib/pds/utils/ZipDecoder.java`
- `/usr/local/kaffe-1.0.6/libraries/javalib/pds/utils/ZipDecoder.spar`
- `/usr/local/kaffe-1.0.6/libraries/javalib/pds/utils/ZipDecoder.java`
- `/users/leo/nl/pds/utils/ZipDecoder.spar`
- `/users/leo/nl/pds/utils/ZipDecoder.java`
- `/users/frits/nl/pds/utils/ZipDecoder.spar`
- `/users/frits/nl/pds/utils/ZipDecoder.java`

If the compilation unit resides in the default package, (it does not have a `package` statement), an empty package name is assumed.

If the compiler now wants to compile a class `ZipDecoder` in the default package, it will try to open the following files in the given order:

[2] This only applies to the *installed* version of the compiler; for the development version another path is used.

- /usr/local/lib/sparlib/ZipDecoder.spar
- /usr/local/lib/sparlib/ZipDecoder.java
- /usr/local/kaffe-1.0.6/libraries/javalib/ZipDecoder.spar
- /usr/local/kaffe-1.0.6/libraries/javalib/ZipDecoder.java
- /users/leo/nl/ZipDecoder.spar
- /users/leo/nl/ZipDecoder.java
- /users/frits/nl/ZipDecoder.spar
- /users/frits/nl/ZipDecoder.java

Note that searching for a class in the default package rarely happens: a class or interface in the default package is only sought if a compilation unit in the default package refers to an unknown class or interface with an unqualified name.

A.2.2 Spar features are only supported in `.spar` files

The Timber compiler only supports the Spar language extensions in files that have the suffix `.spar`. For files with the suffix `.java` of `.jav`, only the standard Java language constructs are supported. In particular, the additional keywords used in Spar (such as `inline` and `foreach`) are only recognized in `.spar` files. In `.java` files they can be used as ordinary variables.

A.2.3 Command-line options of the `spar` script

The Spar script supports the following options (there are more):

`--help`	Show a help text.
`--keepfiles`	Keep intermediate files.
`--nocards`	Do not allow cardinality lists.
`--nocomplex`	Do not recognize the keyword `complex`.
`--nodelete`	Do not recognize the keyword `__delete`.
`--noeach`	Do not recognize the keywords `each` and `foreach`.
`--noinline`	Do not recognize the keyword `inline`.
`--noinlining`	Do not automatically inline any methods or constructors.
`--nopragma`	Do not allow pragmas.
`--strictanalysis`	Analyze definite assignment exactly as in Java.
`--noprint`	Do not recognize the `__print` and `__println` keywords.
`--java`	Do not recognize any Spar extensions.
`--java-array`	Only allow Java array declarations.
`-h`	Show a help text.
`-o <file>`	Write output (executable) to the given file.

The flag `--strictanalysis` specifies that analysis of definite assignment of variables as described in JLS2 §16 is strictly adhered to. Without this flag the compiler does a more sophisticated analysis of the program, and may conclude that a variable is always assigned to before it is used, where a strictly adhering compiler would report that the variable may be used before being assigned.

For example, JLS2 Chapter 16 specifies that the code

```
{
    int k;
    int n = 5;
    if (n > 2)
        k = 3;
    System.out.println(k);
}
```

must cause a compile-time error, since `k` may be used before it is assigned. By default, the Timber compiler will not produce this error message, since it knows that the assignment `k = 3` is always executed. By specifying the the `--strictanalysis` flag, the compiler will adhere to strict Java analysis semantics, and will produce the error message.

Appendix B

Restrictions and incompatibilities of version 1.5 of the Timber compiler

In this appendix we describe the restrictions and incompatibilities of version 1.5 of the Timber compiler. The restrictions and incompatibilities in this appendix are caused by the compiler, and are not a fundamental consequence of extending Java with the extensions of Spar. Thus, a different compiler for Spar/Java does not necessarily have the same restrictions and incompatibilities.

As much as possible, this appendix follows the order of the Java Language Specification version 2 [7] is followed.

B.1 Types, values and variables

B.1.1 The `__string` primitive type

As an extension to Java, the Timber compiler supports the primitive type `__string`. This type is mainly intended as a light-weight representation of string constants. It directly maps to the strings in the Vnus intermediate representation.

There is an implicit widening conversion from `__string` to `String`.

Operations on `__strings`

The Timber compiler provides a number of operators that act on `__strings`:

- The numerical equality operators `==` and `!=`; they result in a value of type `boolean`.
- The string concatenation operator `+`, which, when given two `__string` constants, creates a new `__string` constant.
- The string concatenation operator `+`, which, when given a `String` operand and a `__string` operand, will convert the `__string` operand to a `String`

through internalization, (see JLS2 3.10.5), and then produces a newly created `String` by concatenating the two strings.

For example, the program[1]:

```
public class vnusstring {
    public static void main(){
        __string s = "Hello "+"world";
        __println( 1, s );
    }
}
```

generates the following output:

```
Hello world
```

B.1.2 Unicode

The Timber compiler supports Unicode encoding of string and character constants, but currently characters are stored as 8-bit values.

B.2 Conversions and promotions

B.2.1 Kinds of Conversion

Widening Reference Conversions

JLS2 5.1.4 describes conversions from array types to types `Object`. `Clonable`, and `java.io.Serializable`. Although these conversions are accepted, they are in fact not implemented, and cause an error later in the compilation process.

In addition to the widening reference conversions of JLS2 5.1.4, the Timber compiler supports the following *widening reference conversions*:

- From a `__string` to type `String`.
- From a `__string` to type `Object`.

Both conversions cause the value to be converted to a `String` through internalization (see JLS2 3.10.5).

Narrowing Reference Conversions

Although conversions from type `Object` to array types are accepted, they are in fact not implemented, and cause an error later in the compilation process.

B.2.2 Casting conversion

Casts between classes are currently not checked at run-time. This causes the Timber compiler to accept conversions that would fail in Java. See JLS2 5.5 for a description of casting conversion.

[1] See §B.9.4 for an explanation of the `__println` statement.

B.3 Classes

B.3.1 `transient` and `volatile` fields

Although a field may be marked `transient` or `volatile`, this information is not used in any way by the compiler.

B.3.2 `synchronized` and `strictfp` methods

Although a method may be marked `synchronized` or `strictfp`, this information is not used in any way by the compiler.

B.4 Interfaces

B.4.1 `transient` and `volatile` fields

Although a field may be marked `transient` or `volatile`, this information is not used in any way by the compiler.

B.4.2 `synchronized` and `strictfp` methods

Although a method may be marked `synchronized` or `strictfp`, this information is not used in any way by the compiler.

B.5 Arrays

In Java, arrays are treated as special subclasses of the class `Object`, but in the current Timber compiler they are treated as independent types. This has a couple of consequences:

- Casts from arrays to `Object` and vice versa are accepted by the compiler, but cause an internal error.
- The `clone()`, `toString()`, `equals()`, and `getClass()` methods on arrays are not supported.

B.6 Exceptions

B.6.1 Compile-Time Checking of Exceptions

Compile-time checking of Exceptions, as described in JLS2 11.2, is currently only partially implemented. It *is* checked whether a method or constructor only throws checked exceptions that are permitted by the `throws` clause of that method or constructor.

What is *not* yet enforced is that the `throws` clause of an overriding method cannot allow a wider range of checked exceptions than the method it overrides.

B.6.2 Standard Runtime Exceptions

The following exceptions are never thrown, although specified in the Java Language Specification:

- `ArithmeticException`. In the current Timber compiler, the behavior of the program is unspecified.
- `ArrayStoreException`. In the current Timber compiler, such violations are never detected at run-time.
- `ClassCastException`. Currently all casts are accepted.

B.6.3 Loading and Linkage exceptions

See JLS2 11.5.2. Because of the nature of the current Timber compiler, the exceptions `ClassCircularityError`, `NoClassDefFound`, `IllegalAccessError`, `InstantiationError`, `NoSuchFieldError`, `NoSuchMethodError`, `ClassChangeError`, `VerifyError`, and `AbstractMethodError` are never thrown.

The `ExceptionInInitializerError` is never thrown, instead the original exception is propagated.

B.6.4 Virtual Machine Errors

See JLS2 11.5.2. The exceptions `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError` are never thrown.

B.7 Execution

Since Timber is a static whole-program compiler that directly generates machine code, a number of JLS2 sections on virtual machine behavior do not apply.

Initialization of classes and interfaces, and creation of new class instances is done according to the Java language specification.

B.7.1 Virtual Machine Start-Up

The Timber implementation does not have a virtual machine to start up.

B.7.2 Loading of Classes and Interfaces

Conceptually, the Timber implementation has loaded all class methods and constructors that may be reachable as far as a static compiler can determine. During execution of the program, it is not allowed to load new classes or interfaces.

B.7.3 Linking of Classes and Interfaces

Conceptually, the Timber implementation already performs linking during the compilation phase. Since no `.class` files are ever referenced, no verification of the binary representation is ever done.

B.8 Binary Compatibility

B.9 Blocks and Statements

B.9.1 The `switch` statement

In contrast to standard Java, in the current Timber compiler each *SwitchBlockStatementGroup* is considered a separate block. This means that in contrast to Java variable declarations are not visible in the *SwitchBlockStatementGroup*s below it. For example, the following code is correct in Java, but not in Timber:

```
int n = 3;
switch( n ){
    case 0:
        int x = 42;
        break;

    case 1:
        x++;      // For Timber, x is not visible here.

    default:
        break;
}
```

This restriction is not likely to change in the near future. The most important reason is that translating this correctly would have a significant impact on the entire compilation path. Considering the extreme ugliness of this construct, this restriction may be considered a feature instead of a bug.

B.9.2 The `synchronized` statement

Although the `synchronized` statement is recognized, it is translated as if it is a simple statement block. No locking is performed.

B.9.3 The `try` statement

Timber currently does not execute a `finally` clause when it executes a `break` statement. For example, the following program:

```
public class breakfinally {
    public static void main( String args[] ){
        System.out.println( "Start" );
outer:
        for( int j=0; j<3; j++ ){
            try {
                for( int i=0; i<3; i++ ){
                    System.out.println( "i="+i+" j="+j );
                    if( i == 1 && j == 1 ){
                        break outer;
                    }
                }
```

```
            }
            finally {
                System.out.println( "Hello from finally" );
            }
        }
        System.out.println( "Stop" );
    }
}
```

will produce the following output:

```
Start
i=0 j=0
i=1 j=0
i=2 j=0
Hello from finally
i=0 j=1
i=1 j=1
Stop
```

In the Java version a second line "Hello from finally" would be printed, just before the line with "Stop", since when a `break` passes a `finally` clause, it should execute that clause.

B.9.4 The debugging print statements

For development and testing purposes, the Spar compiler supports the `__print` and `__println` statements, which have the following syntax:

> *Statement*:
> `__print` (*ArgumentList*) ;
> `__println` (*ArgumentList*) ;

The first argument in the list must be of type `int`. This argument denotes the output stream: `1` for the standard output stream, or `2` for the standard error stream. The behavior for other values is unspecified.

The remaining arguments should be of a primitive type[2]. A textual representation of these arguments is written to the given stream.

In the `__println` statement, the string `"\n"` is written to the output stream after all arguments have been written.

For example, the following program:

```
public class vnushello {
    public static void main(){
        __print( 1, "Hello " );
        __println( 1, 1, ",", true, " world" );
    }
}
```

will produce the following output:

[2]Remember that string constants are of type `__string`, and hence are of a primitive type. Type `String`, on the other hand, is not a primitive type.

```
Hello 1,TRUE world
```

The use of these statements is not recommended. They are only documented here because, for performance reasons, they are used in many examples and tests.

B.9.5 The __delete statement

The __delete statement tells the runtime system to delete the given element. For example:

```
public class delete {
    public static void main(){
        short b[*,*] = new short[6,8];
        int c[*] = new int[] { 1, 2, 3, 4 };

        // ... Operations go here ...

        __delete b;
        __delete c;
    }
}
```

The __delete statement is an internal statement, and should not be used in ordinary programs. It is only listed for completeness.

B.10 Expressions

B.10.1 FP-strict Expressions

In contrast to the behavior described in JLS2 15.4, the current Spar implementation does not support FP-strict evaluation.

B.10.2 Expressions and Run-Time checks

In contrast to the behavior described in JLS2 15.5, Spar does not currently do any run-time checks on the correctness of a type. This means that if in the execution of a Spar a run-time check is required, the program will behave differently from a Java program:

- In a cast it is not checked whether the actual source type is compatible with the target type specified in the cast expression. Instead, such a cast is always permitted.
- In an assignment to an array component of reference type, no checking is done.

B.10.3 Normal and Abrupt Completion of Evaluation

Currently most of the causes for abrupt termination that are listed in JLS2 15.6 do not lead to the behavior specified there. Instead, the behavior of the program is unspecified. In particular, in some cases it may lead to abrupt termination of the program, in other cases evaluation of the expression may complete normally, perhaps something else happens.

B.10.4 Primary Expressions

Class Literals

The expression `<type>.class`, as described in JLS2 15.8.2, is correctly parsed, but only partially supported. For primitive types the `TYPE` field of the corresponding wrapper class is accessed, but these currently contain a `null` pointer. For example, `int.class` is replaced by the expression `Int.TYPE`. The current compiler initializes this field to `null`, a future compiler will put something useful in it.

B.11 Definite assignment

Definite assignment, as described in JLS 16, is mostly implemented. There is one particular case where the current compiler deviates from the JLS, namely in `switch` statements. For example, with a switch statement like this:

```
int x;
switch( n ){
    case 0: x = 4; break;
    default: x = 2; break;

}
```

The Spar compiler will not consider `x` to be definitely assigned, since it analyzes each `switch` case separately. However, the JLS proscribes that after execution of a `switch` statement like this `x` is definitely assigned, see JLS 16.2.7.

Another flaw of the current implementation is that the `each` statement is not analyzed properly. For example:

```
int x, y;
each {
    x = 1;
    y = x;
}
```

should cause an error, since it is not guaranteed that `x` has a value at the moment that it is used as a value for `y`. A similar test must be implemented for the `foreach` statement.

It has not yet been verified whether the Spar compiler complies with the definition of definite assignment in the second edition of the JLS.

B.12 Threads and Locks

Threads and locks, as described in JLS2 17, are not implemented.

B.13 The standard library

In the current Timber run-time environment, the method `java.lang.Complex.valueOf(String)`, and the constructor `java.lang.Complex.Complex(String)` are not yet implemented.

Appendix C

Planned extensions and modifications

C.1 Tuples and vectors

- Range operations such as `v[:] = 0;`.

C.2 More conversions to and from tuples

To improve backward compatibility with Java, it is useful to define conversions between 1-tuples and scalars.

For convenience it is useful to allow a complex number to be cast to a `[double^2]` tuple and vice versa. As a further extension, tuple matching on complex numbers should be supported. For example:

```
double a, b;
complex c = 12+3i;

[a,b] = c;
```

Would result in `a` having the value 12, and `b` having the value 3.

C.3 Array expressions

An array expression is a shorthand notation for the construction of a (partial) copy of a given array. For example, the following code will first construct an array `a`, and then construct a copy of the first row of `a`, and assign it to `v`.

```
int[*,*] a = {{0,1,2},{3,4,5},{6,7,8}};
int[*] v = a[0,0:a.getSize(1)];
```

Note that `v` is a *copy* of a part of `a`. Subsequent assignments to elements of `v` will not be visible in `a`. Also note that contrary to array range notations in other languages, the top of the specified range is the first element *not* to be included.

The usual range shorthands apply: if no start of the range is given, 0 is assumed, and if no end of the range is given, the size in that dimension is assumed. Thus the declaration of v in the previous code fragment could be written as:

```
int[*] v = a[0,:];
```

C.4 Array statements

Array statements are a shorthand notation for a foreach statement that is executed for all elements of a selected range. For example, the code fragment

```
Block[*,*] a = new Block[5,7];
a[:,:].init();
```

will invoke the method init on all elements of Block array a.

Note that since this is equivalent to a foreach statement, the init method of each of the array elements is not invoked in a prescribed order. See section 10.2 for more details.

In a similar way array assignments are a shorthand for repeated assignments. For example:

```
int[*,*] a = new int[5,7];

a[:,:] = 0;
```

will zero the entire array a.

The expression at the right-hand side of the assignment will be evaluated only once. Thus,

```
int ix = 0;
int[*,*] a = new int[5,7];

a[:,:] = ix++;
```

will again zero the entire array a, and will leave ix with the value 1.

Last but not least, array assignments may contain an array at the right-hand side, instead of a single element. In that case every iteration of the foreach will use the implicit iteration variable as index for every assignment.

For example:

```
int ix = 0;
int[*,*] a = new int[5,7];
int[*,*] b = new int a.getSize();

b[:,:] = 1;
a[:,:] = b;
```

Will copy b into a. The last statement could also be written as:

```
a[:,:] = b[:,:];
```

but a naive compiler would first create a copy of b, and leave it for the garbage collector.

Array statements never change the size of the array they work on. Any access that is out of bounds is detected at compile-time or run-time, and causes an error message or an IndexOutOfBoundsException exception.

Since array statements are shorthands for foreach statements, a Spar compiler will probably expand them to the appropriate foreach early in the compilation process. This may burden the task mapper with iteration bodies that are too small for meaningful mapping. I assume that the task mapper is smart enough not to map such tasks.

This interpretation of array statements is consistent, easy to understand and easy to implement, but it is not environmentally friendly: it leaves lots of garbage. With some analysis, we can prevent actually constructing the array slice, in many, but not all cases. For example, f(A) and f(A[:]) are different, because the latter will pass a copy of A. This means that we must rely on an optimizer for efficient code generation.

Also, the lack of symmetry between right-hand side and left-hand side array expressions is ugly.

The following code shows some array statements.

```
int[*] a = [0,1,2,3,4,5,6,7];
int[*] b = new int[8];

b[0:3] = a[1:4];
b[:3] = a[1:4];          // Identical to previous statement
b[:] = a[:];             // All elements are copied.
```

But this is *not* an array statement; it merely copies a reference:

```
b = a;
```

Appendix D

Supported pragmas

In this appendix we describe the pragmas that are currently supported by the compiler. For each of these pragmas a grammar is given for the allowed values of the pragma. These grammars describe a strict subset of the general pragma value syntax. That is, all valid values for these specific grammars are also valid values for the general syntax, but reverse is not true. In reality the additional restrictions are not enforced by a parser, but by the engines that handle these pragmas.

The grammars for the pragmas use a minor extension to the grammar notation described in JLS2 2.4. The subscripted suffix "*seq*", which may appear after a terminal or nonterminal, indicates a *sequence* of symbols. A symbol sequence represents the juxtaposition of an arbitrary number of symbol instances, without any separators inbetween. For example, grammar rule for a pragma list expression could be expressed as:

PragmaExpression:
  `(` $\mathit{PragmaExpression}_{seq}$ `)`

See also section 12.

D.1 Pragmas for parallelization

D.1.1 The `ProcessorType` pragma

The `ProcessorType` pragma is used to declare a processor type and to describe the processors characteristics (e.g. alignment of data structures and endianness of primitive types etc.) and capabilities (e.g. whether it contains a FPU etc.). `ProcessorType` pragmas must be global.

By making the processor types explicitly visible to the compiler, they can be used as a sort of type checking mechanism. For instance, if a floating point operation is mapped onto a DSP which does not support IEEE floating point operations, the compiler can signal a placement error to the user. They can also be used to avoid excessive code size by selectively compiling member functions.

A `ProcessorType` pragma should be written as follows:

ProcessorType-Pragma:
  `ProcessorType = (` $\mathit{ProcessorTypeSpec}_{seq}$ `)`

ProcessorTypeSpec:
`(` *ProcessorType-Identifier Resource-StringLiteral* `)`

The *ProcessorType-Identifier* is the name of the processor type. The *Resource-StringLiteral* specifies the resource where details about the processor can be found. For example:

```
<$ ProcessorType=((Gpp "Pentium2") (Dsp "Trimedia")) $>
```

The strings `"Pentium2"` and `"Trimedia"` refer to configuration specifications of the processors. The way the specification is stored, and format of the specification, is left to the implementation.

D.1.2 The `Processors` pragma

A `Processors` pragma is used to name and list the processors in a system. The `Processors` pragma must be a global pragma. Currently only a single `Processors` pragma is allowed in a program; multiple 'views' on the hardware, like in HPF [10], are not allowed. This greatly simplifies the compile-time analysis, since it avoids alias analysis on processor locations.

A `Processors` pragma should be written as follows:

Processors-Pragma:
`Processors = (` *Processor-Identifier*$_{seq}$ `)`

Processor:
`(` *Processor-Identifier ProcessorIdentifier* `)`

ProcessorIdentifier:
ProcessorType-Identifier
`( at` *ProcessorType-Identifier IntLiteral*$_{seq}$ `)`

Each processor type used in the system must have been declared with a `ProcessorType` pragma. You can declare either a single processor or an array of processors. The processor array can have any number of dimensions, but the size in each dimension must be an integer literal. For example[1]:

```
<$ Processors=(( Gpp gpp1) (Dsp dsp1D[4]) (Dsp dsp2D[2,3])) $>
```

The system specified above consists of a single processor `gpp1` of type `Gpp`, a one-dimensional processor array `dsp1D`, and a two dimensional array `dsp2D`, both of type `Dsp`. Each modeled processor corresponds to a single physical processor.

D.1.3 The `on` pragma

A Spar/Java program can be annotated at specific points with an `on` pragma that allows users to place data and work on specific processors. They may be nested and are completely independent of the enclosing `on` pragma specification. They can annotate expressions, statements, and member functions. An `on` pragma nested inside another `on` pragma overrules the enclosing specification.

The specific details of code generation for each of the processors, and the synchronization amongst processor tasks and communication between processors are left to the compiler.

The syntax of the `on` pragma is:

[1] We use syntactic sugar for the `(at ...)` expression, see §12.1.

on-Pragma:
`on =` *ProcessorReference*

ProcessorReference:
Processor-Identifier
`( at` *Processor-Identifier* *Index-PragmaExpression*$_{seq}$ `)`

IndexExpression:
IntegerExpression
PlacementFunction

PlacementFunction:
`( block` *Integer-PragmaExpression* *BlockSize-PragmaExpression* `)`
`( cyclic` *Integer-PragmaExpression* *BlockSize-PragmaExpression* `)`

An `on` pragma consists of a *ProcessorReference*. A *ProcessorReference* consists of a *Processor-Identifier* as defined in the `Processors` pragma, or an `at` list containing a *Processor-Identifier*, followed by zero or more *Index-PragmaExpression*s indexing a processor array. The special *Processor-Identifier* `_all` is used to denote all processors; the special *Processor-Identifier* '_' (a single underscore) is used to denote an unspecified placement ('don't care').

Every *Index-PragmaExpression* in an `on` pragma must evaluate to an integer expression. It is explicitly allowed to refer to integer variables in the host program through an `@` expression (see §12). The special identifier '_' leaves the placement in the corresponding processor dimension unspecified ('don't care'). The special identifier `_all` specifies a placement on all processors in the corresponding dimension of the processor array.

It is also allowed to use two special functions as an *Index-PragmaExpression*: The `(block i m)` placement function places index i onto processor $p = i/m$. The value of p is bounded by the index range allowed in the corresponding dimension of the processor type (that is, $0 \leq p < P_{ext}$). If no m is specified, the value is derived from the context: if there are N elements in the corresponding array dimension, or if there are N iterations in the corresponding iteration range, $m = N/P_{ext}$ is assumed. The `(cyclic i m)` placement function places index i onto processor $p = (i/m) \bmod P_{ext}$. If no m is specified, the value 1 is assumed.

For data that is distributed with the `block` and `cyclic` functions, the compiler is able to apply specific optimizations, see [17] for details. With these functions, all HPF data mappings can be specified, even alignments, although templates are not explicitly visible as in HPF.

Annotating declarations

By annotating a member function, the user can specify the group of processors allowed to execute the member function. For example:

```
<$ on dsp1D[@i] $> int myfunc( int i, String s ) { return i+1; }
```

Annotating statements

A statement annotated with an `on` pragma will be executed only on the specified processors. In principle, arbitrary statements may be annotated, but in practice the annotation is only interesting for code *blocks*. For example:

```
foreach( i :- 0:100 ) <$ on=dsp1D[(block @i 25)] $> { a[i] = a[i] + 1; }
```

The assignment of `a[i]` is executed on processor `dsp1[i/25]`. In other words, the complete iteration space is divided into blocks of size 25, and each block is executed by a different processor.

Annotating expressions

In principle, any expression can be annotated with an `on` pragma. This is especially useful for a `new` expression, since this not only specifies the placement of the constructor execution (if not overridden by an annotation on the constructor), but also the placement of the newly constructed class or array instance. For example, the pragma:

```
String a = <$ on=gpp1 $> new String();
```

specifies that a new `String` must be constructed on `gpp1`.

In the case of array `new` expressions, a slightly extended version of the `on` pragma is allowed:

on-Pragma:
  `on =` *ProcessorReference*
  `on =` *AbstractProcessorReference*

AbstractProcessorReference:
  `( lambda (` *Identifier*$_{seq}$ `)` *ProcessorReference* `)`

The *AbstractProcessorReference* defines a function that places each element of the newly constructed array to a processor. In other words, such a function defines the *distribution* of the new array.

The sequence of *Identifier*s contains the formal parameters of the placement function. The sequence may not contain duplicates, and must contain as many elements as the rank of the array. All formal parameters are implicitly of type `int`. For example, the pragma:

```
int[*,*] b = <$ on=(lambda (i j) dsp2D[(block j 5),_all]) $> new int[50,50];
```

specifies that every array element `b[i,j]` is constructed on processors `dsp2D[(block j 5),_all]`. The `_all` expression in the second dimension means that the elements are *replicated* in the second dimension of the processor array.

Note that since the formal parameter `i` is not used in the distribution expression, the first dimension of the array does not influence the distribution of an element.

D.2 Pragmas for optimization

D.2.1 The `inline` pragma

A method or constructor that is labeled with the `inline` pragma will be declared `inline` in the generated C++ code. This annotation is separate from the `inline` *modifier*, which requests the `Spar` compiler to inline the method or constructor. Thus, a method

```
inline max( int a, int b ){ return a>b?a:b; }
```

Will be inlined, if possible, by the Spar compiler. A method

```
<$ inline $> max( int a, int b ){ return a>b?a:b; }
```

will be inlined, if possible, by the C++ compiler.

The compiler will annotate a method with the `inline` modifier with the `inline` pragma.

Bibliography

[1] R. T. Boute. Funmath: towards a general formalism for system description in engineering applications. In P. P. Silvester, editor, *Advances in Electrical Engineering Software*, pages 215–226. Computational Mechanics Publications, Southampton, and Springer-Verlag, Berlin, August 1990.

[2] Mary Campione and Kathy Walrath. *The Java Tutorial*. The Java Series. Addison-Wesley, Reading, Massachusetts, August 1996.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical report, California Institue of Technology, September 1992.

[4] A. González Escribano, Valentín Cardeñoso Payo, and A.J.C. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proc. 1st EURO-PDS Int'l Conf. on Parallel and Distributed Systems*, Barcelona, June 1997.

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

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, Massachusetts, June 2000.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, Massachusetts, June 2000.

[9] R. W. Harper, D. B. MacQueen, and R. Milner. Standard ML. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1986. Also CSR-209-86.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, February 1997.

[11] Paul Hudak. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[12] ISO/IEC. *ISO/IEC 1539 (Fortran 90)*, second edition, July 1991.

[13] David May. Occam. In *IFIP Conference: System Implementation Languages: Experience and Assessment, Canterbury*, September 1984.

[14] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997. ACM.

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

[16] C. van Reeuwijk. *The implementation of a system description language and its semantic functions.* PhD thesis, Delft University of Technology, Department of Electrical Engineering, Delft, The Netherlands, July 1991.

[17] C. van Reeuwijk, W. Denissen, H.J. Sips, and E.M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, September 1996.

[18] David Turner. Miranda: A non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional programming languages and computer architecture*, volume 201 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[19] David Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.

Index