The Deterministic Parallel Java Language Specification Version 1.0

University of Illinois at Urbana-Champaign

DRAFT: Revised May 2010

This document describes Deterministic Parallel Java (DPJ) as an extension to the Java Programming Language, v1.6 (Java 6). For the specification of Java 6, see The Java Programming Language Specification [3], hereafter referred to as "JLS."

The specification proceeds as follows. In §1, we describe *region path lists*, or RPLs, which provide a way to name sets of memory locations on the heap in a DPJ program. In §2, we describe *regions*, which logically partition memory and allow the specification and checking of effects. There are heap regions and stack regions. RPLs name sets of heap regions, while stack regions are automatically managed by the compiler. In §3, we describe the DPJ extensions to Java's class and interface definitions. In §4, we describe the DPJ extensions to Java's class and array types. In §5 we describe *effects*, which specify accesses to memory in terms of operations on regions. In §6, we describe DPJ's constructs for expressing parallelism.

Contents

1 Region Path Lists

In DPJ, the programmer uses region path lists, or RPLs, to name sets of heap regions (§2.1). The region sets in turn are used in specifying and checking effects (§5).

1.1 Valid RPLs

A region path list (RPL) is a nonempty sequence of RPL elements (§1.2) separated by colons (`:`). An RPL is well-formed if it obeys the following rules:

- The first element must be `Root`, a local region name (§1.2.1), an RPL parameter element (§1.2.2), or a variable element (§1.2.4).
- `Root`, an RPL parameter element, or a variable element may appear only as the first element.

It is also acceptable to write an RPL starting with a class region name element (§1.2.1). In this case, the RPL has `Root` as an implicit first element.

1.2 RPL Elements

An RPL element is one of the following: a name RPL element, a parameter RPL element, an array index RPL element, a variable RPL element, or the star RPL element.

1.2.1 Name RPL Elements

A name RPL element is one of the following:

- `Root`, which is a reserved name available at global scope.
- A class region name declared in the enclosing class scope (§3.2).
- A local region name declared in the enclosing statement scope (§3.4.4).
- *selector*`.`*region*, where *selector* is a field access selector (i.e., everything to the left of `.`*Identifier* in a field access expression, JLS §15.11), and *region* is a class region declared in the class named by *selector* (§3.2).

1.2.2 Parameter RPL Elements

A parameter RPL element names a class RPL parameter (§3.1), method RPL parameter (§3.4.1), or capture parameter (§4.1.8) that is in scope. Note that capture parameters are introduced by the compiler in capturing types (§4.1.8), and cannot be written by the programmer.

1.2.3 Array Index RPL Elements

An array index RPL element is one of the following:

- `[`*expr*`]`, where *expr* is a valid integer expression, indicating the array index corresponding to the runtime value of *expr*.
- `[?]`, indicating any array index.

1.2.4 Variable RPL Elements

A variable RPL element is one of the following: `this`; a `final` method variable of class type; or a `final` method formal parameter of class type.

1.2.5 The Star RPL Element

The star RPL element has the form `*`. It indicates any sequence of RPL elements.

1.3 Relations on RPLs

RPLs have several pairwise relations that are used to determine when two RPLs name disjoint or overlapping sets of regions: equivalence (§1.3.1), nesting (§1.3.2), disjointness (§1.3.4), and inclusion (§1.3.3).

1.3.1 Equivalence

There is an equivalence relation on RPLs, which we will denote here by "*rpl-1* is equivalent to *rpl-2*." Its meaning is that the set of regions represented by *rpl-1* is equal to the set of regions represented by *rpl-2*.
Equivalent RPLs: Two RPLs are equivalent if both RPLs have the same number of elements, and the corresponding pairs of elements are equivalent.
Equivalent RPL Elements: Two RPL elements are equivalent if:

- They refer to the declared same region name, RPL parameter, or variable name; or
- They are `[`*expr-1*`]` and `[`*expr-2*`]`, where *expr-1* and *expr-2* are always-equal expressions; or
- They are both `[?]` or `*`.

Always-Equal Expressions: Always-equal expressions are defined as follows:

- Two constants are always-equal expressions if they are the same value.

- Two variables are always-equal expressions if they are the same variable (i.e., they have the same variable symbol) and the variable is declared `final`.
- Two binary operation expressions are always-equal expressions if they represent the same operation, and their corresponding component expressions are always-equal.
- Any other two expressions are not always-equal.

1.3.2 Nesting

There is a nesting relation on RPLs, which we will denote here by "*rpl-1* is nested under *rpl-2*." Its meaning is that for every region *r-1* represented by *rpl-1*, there exists a region *r-2* represented by *rpl-2* such that *r-1* is nested under *r-2*. The meaning of "*r-1* is nested under *r-2*" Is that both *r-1* and *r-2* are part of the same tree of regions, and *r-1* is a descendant of *r-2* in the tree. *rpl-1* is nested under *rpl-2* if one of the following holds:

- *rpl-2* is `Root`.
- *rpl-1* is a sequence of two or more RPL elements, and *rpl-1* without its last element is nested under *rpl-2*.
- *rpl-1* is included in *rpl-2* (§1.3.3).
- *rpl-1* has one element, which is an object reference variable whose owner RPL (§4.1.7) is nested under *rpl-2*.

1.3.3 Inclusion

There is an inclusion relation on RPLs, which we will denote here by "*rpl-1* is included in *rpl-2*." Its meaning is that the set of regions represented by *rpl-1* is contained (in the sense of ordinary set inclusion) in the set of regions represented by *rpl-2*.
Inclusion of RPLs: *rpl-1* is included in *rpl-2* if one of the following holds:

- *rpl-1* and *rpl-2* are equivalent (§1.3.1).
- *rpl-2* ends with `*`, and *rpl-1* is nested under *rpl-2* (§1.3.2) without its last element.
- The last element of *rpl-1* is included in the last element of *rpl-2*, and inclusion holds for the RPLs after stripping the last elements of both.
- *rpl-1* has one element, which is a capture parameter (§4.1.8) included in *rpl-2*.

Inclusion of RPL Elements: RPL element *elt-1* is included in RPL element *elt-2* if (1) the elements are equivalent (§1.3.1); or (2) *elt-1* is any array index RPL element and *elt-2* is `[?]`.

1.3.4 Disjointness

There is a disjointness relation on RPLs, which we will denote here by "*rpl-1* is disjoint from *rpl-2*." Its meaning is that the set of regions represented by *rpl-1* has nonempty intersection with the set of regions represented by *rpl-2*.
Disjoint RPLs: *rpl-1* and *rpl-2* are disjoint if one of the following holds:

1. *rpl-1* is included in (§1.3.3) some RPL *rpl-3*; *rpl-2* is included in some RPL *rpl-4*; and *rpl-3* and *rpl-4* are constrained to be disjoint (§§3.1 and 3.4.1).
2. One of *rpl-1* and *rpl-2* is nested under (§1.3.2) an RPL that starts with a local region name (§3.4.4), and the other is not.
3. For some $n \geq 1$, (a) the RPLs formed by taking the first n elements of *rpl-1* and *rpl-2* are equivalent (§1.3.1); and (b) *rpl-1* and *rpl-2* have disjoint elements in position $n + 1$ (this is called a "distinction from the left" in [2]).
4. The last elements of *rpl-1* and *rpl-2* are disjoint (this is called a "distinction from the right" in [2]).

Disjoint RPL Elements: Two RPL elements *elt-1* and *elt-2* are disjoint if (1) one is a name RPL element (§1.2.1) and the other is an array index RPL element (§1.2.3); (2) they are both name RPL elements corresponding to distinct region name symbols (§§3.2 and 3.4.4); or (3) they are the array RPL elements `[`*expr-1*`]` and `[`*expr-2*`]`, where *expr-1* and *expr-2* are always-unequal expressions.
Always-Unequal Expressions: Always-unequal expressions are defined as follows:

- Two constants are always-unequal expressions if they are different values.
- *expr-1* and *expr-2* are always-unequal expressions if *expr-1* is the negation expression (§6.2) of *expr-2*, or vice versa.
- Any other two expressions are not always-unequal.

2 Regions

DPJ partitions memory into *heap regions* (collections of memory locations on the heap) and *stack regions* (collections of memory locations on the stack). Heap regions are explicitly named using RPLs (§1). Stack regions are internally named and automatically managed by the compiler.

2.1 Heap Regions

A heap region names a set of memory locations on the heap at runtime. Syntactically, a heap region is like an RPL (§1), with the following exceptions:

- No `*` or `[?]` appears in the sequence of elements.
- There are no parameter RPL elements.
- Object reference values appear instead of object reference variables.
- Integer values `[`n`]` appear instead of integer expressions `[`*expr*`]`.

An RPL with no `*` or `[?]` corresponds to a heap region at runtime, by replacing parameters with actual regions, object reference variables with the references they store, and integer expressions with the values to which they evaluate. For more information about how this substitution works, see [1]. An RPL containing `*` or `[?]` represents the set of valid RPLs obtained by substituting (possibly empty) sequences of RPL elements for `*` and nonnegative array index values for `?`. This set of RPLs in turn corresponds to a set of regions, via the substitutions mentioned above.

Every heap region is part of a tree of regions. There is one global tree rooted at `Root`. Inside a method body, there is also one local tree for every declared local region name (§3.4.4) in scope. The tree structure is given in two ways:

1. By the syntax of RPLs: e.g., `A:B` is a child of `A`.
2. By the types of object references: if o is an object reference, and the owner region of the type of o (§4.1.7) is R, then a region whose RPL starts with o is a descendant of region R.

For more information about the region tree, see [1].

2.2 Stack Regions

The compiler internally generates *stack regions* for method value parameters and local variables. For example, the declaration `int x` inside a method generates the stack region `x` that is in scope where the variable `x` is in scope. These stack regions are generated and checked automatically, and the programmer cannot specify them directly.

Stack regions are compared by symbol: two occurrences of the same symbol are treated as identical, and different symbols are treated as different. This simple comparison is possible for stack variables because Java does not allow references to or aliasing of these variables.

3 Class and Interface Definitions

3.1 Type and RPL Parameters

As part of a class or interface declaration, the compiler allows a list of generic type and/or RPL parameter declarations to appear directly after the class or interface name, which follows the keyword `class` or `interface`. A parameter declaration extends the generic parameter declaration of Java 1.6 and has the form `<`*params*`>` or `<`*params*`|`*constraints*`>`, where

- *params* is a comma-separated nonempty list of identifiers stating the parameters being declared. The generic type parameters, if any, must come first; followed by the RPL parameters, if any. Any of the type parameters may be preceded by the keyword `type`. At least the first RPL parameter must be preceded by the keyword `region`, and the other RPL parameters may be preceded by the keyword `region`.

- *constraints* is a comma-separated list of *RPL constraints*. An RPL constraint has the form *rpl-1 # rpl-2*, where *rpl-1* and *rpl-2* are legal RPLs (§1.1) in the scope of the class definition.

The type parameters function exactly as in Java 1.6 (notice that if only type parameters appear, and the optional `type` keyword is not used, the syntax corresponds exactly to Java 1.6). The RPL parameters may be used as RPL elements (§1.2) in the scope of the class definition, including in any type constraints appearing in any generic parameter declarations. As with type parameters in Java 1.6, the type and RPL parameters are available only in an instance context (i.e., in any scope where `this` is in scope).

The RPL parameter constraints, if present, impose disjointness constraints (§1.3.4) that are enforced when RPL arguments are bound to the parameters in a type that instantiates the class (§4.1.1). Each constraint *rpl-1 # rpl-2* is also recorded in the environment, so that the compiler can prove that *rpl-3* and *rpl-4* are disjoint, if *rpl-3* is included in (§1.3.3) *rpl-1* and *rpl-4* is included in *rpl-2*. It is an error to write a constraint *rpl-1 # rpl-2* such that *rpl-1* is included in *rpl-2* or vice versa.

3.2 Class Region Name Declarations

A region name declaration may appear as a class member (JLS §8.1.6).

A region name declaration has the form `region` *region-decls*, where *region-decls* is a comma-separated list of identifiers. The declared names are available within the enclosing class scope and via inheritance and/or class selection (as for an ordinary Java static member, JLS §8.2), subject to visibility restrictions. The declared names may be used as name RPL elements (§1.2.1) wherever they are available.

Every region name declaration appearing in a particular class must have a unique name. Otherwise, if two region declarations with the same name appear in the same scope, the innermost declaration hides all the others.

3.3 Fields

As part of a field declaration, the compiler accepts an optional *RPL specifier* with the syntax `in` *rpl*, where *rpl* is a valid RPL (§1.1). If the RPL specifier appears in a field declaration, it must appear after the field name and before the initializer expression, if any. The RPL specifier is meaningful only for non-`final` fields; if an RPL specifier appears in a `final` field, it is allowed but silently ignored. RPL specifiers are not allowed for local variable declarations or formal method parameter definitions, because those variables reside in stack regions (see §5.3).

3.4 Methods

3.4.1 Type and RPL Parameters

As part of a method definition (including constructors), a list of type and RPL parameter declarations may appear inside angle brackets `<...>` in the same position where Java 1.6 allows type parameters, i.e., after the method qualifiers (`public`, `static`, etc.), if any, and before the method return type (for non-constructor methods) or class name (for constructors). A method parameter declaration has the same form as a class parameter declaration (§3.1). The declared RPL parameters may be used as RPL elements in the scope of the method definition, including the formal parameters and return type. The constraints, if present, impose disjointness constraints that are enforced when the parameters become bound in a method invocation (§3.4.2). The constraints are also recorded in the method environment, as described in §3.1 for class parameter constraints.

3.4.2 Type and RPL Arguments

When a method defined with type and/or RPL parameters (§3.4.1) is invoked, the invocation (at the programmer's option) may use either *explicit type and RPL arguments* or *inferred type and RPL arguments*. This approach corresponds to the way that Java 1.6 handles generic method parameters.

Explicit Type and RPL Arguments: As in Java 1.6, if type and/or RPL arguments are present, then there must be an explicit selector (a static class selector or an expression of class or interface type) followed by a dot. The arguments must be enclosed in angle brackets `<...>`, after the dot and before the method name. For constructors, the arguments must follow the keyword `new` and precede the class name. In either case, the RPL arguments have the same form as for class and interface RPL arguments (§4.1.1), except that the keyword `region` must precede at least the first RPL argument. (This requirement tells the compiler which arguments are types and which ones are RPLs. Because of Java's method overloading, this information is not available from the method name, as it is for classes from the class name.) The number of type and RPL arguments must exactly match the number of type and RPL parameters; otherwise, the compiler reports an error.

Inferred Type and RPL Arguments: If a method is defined with type and/or RPL parameters, then the method invocation may be written without explicit type or RPL arguments. The compiler will attempt to infer the arguments from the types of the value arguments, using an algorithm similar to the one that Java 1.6 uses for inferring generic parameters. If the inference is not possible for a given RPL argument (for instance, because the parameter does not appear in any of the actual argument types), then `Root:*` is conservatively inferred for that argument.

RPL Constraints: The compiler checks the RPL arguments against the constraints, as discussed in §4.1.1 for class parameter constraints. If any constraint is violated, the compiler issues a warning.

3.4.3 Method Effect Summaries

Every method may optionally declare its effects via an *effect summary* (§5.2). If an effect summary is present, it must appear after the method arguments, before the method body, and before the `throws` clause, if any. The compiler internally converts each method effect summary to an effect set (§5.1), called the *declared effect set* of the method. If there is no explicit effect summary, then the compiler inserts a *default effect set* containing `writes Root:*`. In particular, an ordinary Java 1.6 method (with no DPJ annotations) is a valid DPJ method, whose effect summary is `writes Root:*`.

The compiler infers the effects of the method body (§5.3) and checks the following, after pruning all effects on RPLs that start with a local region element defined in the method body:

- The inferred effects are a subeffect (§5.5) of the declared effects.
- The declared effects are a subeffect (§5.5) of the declared effects of all methods overridden by this one.

If either of these requirements is not met, the compiler generates an error.

3.4.4 Local Region Name Declarations

A region declaration may appear as a statement in a method body (JLS §8.4.7). The declaration has the same form as for class region name declarations (§3.2). The declared names are available as name RPL elements in the innermost enclosing block.

3.4.5 Commutativity Annotations

The keyword `commutative` may optionally appear as a method qualifier, before the return type and before the method parameters, if any. The `commutative` annotation is a programmer-specified guarantee that the method commutes with itself, i.e., any pair of `invokes` effects (§5.1) using this method are treated as noninterfering. If `commutative` appears in the definition of a method M, then it must also appear in the definitions of all methods overriding M.

When using `commutative`, the programmer is responsible for ensuring two things:

1. The method is properly synchronized so that concurrent invocations of the method behave as though they have occurred in sequence (i.e., the invocations have serializable semantics [4]).

2. The semantics of the method is such that either order of a pair of invocations produces the same result.

Note that “same result” in condition (2) generally has an application-specific meaning. For instance, two insert operations on a set may commute with each other, because they produce the “same” set (i.e., the two sets are indistinguishable as far as future set operations like find are concerned), even though the sets may have different internal representations. However, two append operations on an ordered list generally do not commute.

4 Types

4.1 Class and Interface Types

4.1.1 Type and RPL Arguments

When a class or interface defined with type and/or RPL parameters (§3.1) is used to name a type, the type may optionally specify type and/or RPL arguments.
Syntax: The arguments have the form <*args*>, where *args* is a comma-separated list of valid class or interface types (this section) and/or RPLs (§1). This syntax, if it appears, must immediately follow the class or interface name. The types, if any, must appear first; followed by the RPLs, if any. Any of the types may be preceded by the keyword `type`, and any of the RPLs may be preceded by the keyword `region`; but the keywords are optional, as the compiler can infer which are the types and which are the RPLs from the class or interface definition.
Binding RPL arguments: It is an error to specify more RPL arguments than there are RPL parameters. If fewer RPL arguments than parameters are specified, the remaining arguments (from left to right) become bound to `Root`. For each RPL parameter constraint *rpl-1 # rpl-2* specified in the parameter declaration (§3.1), the compiler checks that the RPLs *rpl-1* and *rpl-2* are disjoint (§1.3.4), after substituting the RPL arguments for the corresponding parameters in the RPLs. If any disjointness constraint is violated, the compiler issues a warning.

4.1.2 Equivalent Types

Let T_1 and T_2 be class or interface types. T_1 and T_2 are equivalent if (1) they instantiate the same class or interface C; (2) for every type parameter P of C, the type arguments bound to P in T_1 and T_2 are equivalent (this section); and (3) for every RPL parameter P of C, the RPL arguments bound to P in T_1 and T_2 are equivalent (§1.3.1).

4.1.3 Inclusion of Types

Let T_1 and T_2 be class or interface types. T_1 is included in T_2 if (1) T_1 and T_2 instantiate the same class or interface C; (2) for each type parameter P of C, the type arguments bound to P in T_1 and T_2 are equivalent (§4.1.2); and (3) for each RPL parameter P of C, the RPL argument bound to P in T_1 is included in the argument bound to P in T_2 (§1.3.3).

4.1.4 Superclass Types

Immediate superclass types: Let C be a class or interface definition, and let T_1 be a type instantiating C. If no `extends` or `implements` clause appears in C, then the *immediate superclass type* of T_1 is `Object`. Otherwise, suppose `extends` T_2 appears in the definition of C. Then the type T_3 obtained from T_2, after substituting the actual type and region arguments of T_1 for the type and region parameters of C, is an immediate superclass type of T_1. The same rules apply if `implements` T_2 appears in the definition of C.
Superclass types: Let T and T' be class or interface types. T' is a *superclass type* of T if there exists a nonempty sequence of types $T_1, \ldots, T_n$ such that (1) T_1 is the immediate superclass type of T; (2) T_{i+1} is the immediate superclass type of T_i for all $1 \leq i < n$; and (3) T' is the immediate superclass type of T_n. The *superclass type of T at T'* is the type T'' such that (1) T'' is a superclass type of T and (2) T'' instantiates the same class or interface as T'. If the superclass type of T at T' exists, then it is unique.

4.1.5 Subtypes

Let T_1 and T_2 be class or interface types. We compute whether T_1 is a subtype of T_2 as follows:

1. First, disregard any RPL parameters (i.e., use ordinary Java subtyping). If T_1 is not a subtype of T_2, then the answer is no.
2. If the answer in step 1 was yes, then compute the superclass type of T_1 at T_2 (§4.1.4). Call this type T_3.
3. If T_1 is included in T_3 (§4.1.3), then the answer is yes. Otherwise, the answer is no.

4.1.6 Casts

The compiler disregards RPL parameters in determining whether a cast from one class or interface type to another is legal, so unsound casts are allowed. This is identical to what Java 1.6 does for casting generic types. Any unsound cast will generate the usual "unchecked or unsafe operations" warning that Java 1.6 gives for unsound generic casts.

4.1.7 Owner RPLs

If *class-type* is a class type, then the *owner RPL* of *class-type* is (1) the RPL bound to the first parameter of *class-type*, if *class-type* has parameters; otherwise (2) `Root`.

4.1.8 Captured Types

The compiler extends Java 1.6 type capture (JLS §5.1.10) to capture RPL parameters as well as generic type parameters. If *type* is a class type with *rpls* as its RPL arguments, then the capture of *type* is defined as follows:

1. Take the Java 1.6 capture of *type*, possibly substituting for the generic type arguments but keeping the same RPL arguments.
2. In the result of step 1, for each RPL *rpl* in *rpls* that is partially specified (i.e., contains `*` or `[?]`), replace that RPL with a fresh RPL parameter constrained to be included in *rpl*.

4.2 Array Types

4.2.1 Valid Array Types

In DPJ, the array type is given by *base-type* followed by one or more instances of *brackets*, where *base-type* is a primitive, class, or interface type; and each instance of *brackets* is a pair of brackets (`[]`) optionally followed by an RPL argument <*rpl*>, optionally followed by #*index-var*, where *index-var* is an identifier that declares an index variable. An array type is valid if (1) *base-type* is valid, treating all index variables appearing in the type as integer variables in scope; and (2) the RPL of every *brackets* is valid, treating all index variables appearing in that *brackets* and all *brackets* to the left of that one as integer variables in scope.

4.2.2 New Arrays

A new array expression is `new` followed by *base-type* and one or more instances of *new-brackets*, where *base-type* is a primitive, class, or interface type; and *new-brackets* is the same as *brackets* as described in §4.2.1, except that an integer length expression *expr* may optionally appear between the brackets. A new array expression is valid if (1) the array type generated by deleting the length expressions is valid (§4.2.1); and (2) the first n instances of *new-brackets* contain a length expression, for some $n \geq 1$, while the rest do not.

4.2.3 Subtypes

If T_1 and T_2 are array types, then the compiler checks whether T_1 is a subtype of T_2 as follows:

1. If the RPL specified in the leftmost *brackets* of T_1 is not included in (§1.3.3) the RPL specified in the leftmost *brackets* of T_2, then the answer is no.
2. Otherwise, check whether the element type of T_1 is a subtype of the element type of T_2 (§4.1.5 or this section), but require equivalence of corresponding RPL arguments (§1.3.1), instead of just inclusion, in doing this check.

We require equivalence of RPL arguments for all but the leftmost RPL because it is not sound in general to allow subtype assignments into array cells.

4.2.4 Casts

Array type casts work in the same way as class type casts (§4.1.6).

4.3 Typing Expressions

4.3.1 Field Access

The compiler computes the type of a field access expression *selector-exp*`.`*field-name* as follows: (1) compute the type *selector-type* of *selector-exp*; (2) capture the type (§4.1.8) to generate type *captured-selector-type*; (3) look up the type *field-type* of *field* based on the class C named in *captured-selector-type*; (4) make the following substitutions in *field-type*: (a) the type and RPL arguments of *captured-selector-type* for the type RPL parameters of C; and (b) *selector-rpl* for `this`. If *selector-exp* is a `final` local variable *var*, then *selector-rpl* is *var*; otherwise, *selector-rpl* is *owner-rpl*`:*`, where *owner-rpl* is the owner RPL (§4.1.7) of *captured-selector-type*.

4.3.2 Array Access

If expression e has array type T (§4.2.1), then we construct the type and region of array access expression e`[`e'`]` as follows:

- To construct the type, do the following: (a) concatenate *base-type* of T with all instances of *brackets* in T but the leftmost to form *elt-type*; and (b) if an index variable i is declared in the leftmost instance of *brackets*, then substitute e' for all instances of i appearing in *elt-type*.
- To construct the RPL, do the following: (a) let *rpl* be the RPL appearing in the leftmost instance of *brackets* in T, or `Root` if no RPL appears there; and (b) if an index variable i appears in the leftmost instance of *brackets*, then substitute e' for all instances of i appearing in *rpl*.

4.3.3 Method Invocation

Explicit Type and RPL Arguments: The compiler computes the type of a method invocation expression

selector-exp`.<`*type-args*`,`*rpl-args*`>`*method-name*`(`*args*`)`

as follows: (1) compute the type *selector-type* of *selector-exp* and the types *arg-types* of *args*; (2) capture *selector-type* (§4.1.8) to generate type *captured-selector-type*; (3) look up the method symbol based on the method name, *selector-type*, and *arg-types*, and find the return type *return-type* of the method; (4) make the following substitutions in *return-type*: (a) the type and RPL arguments of *captured-selector-type* for the type and RPL parameters of *return-type*; (b) *type-args* and *rpl-args* for the type and RPL parameters of the method; (c) for every expression *int-exp* in *args* of a type assignable to `int`, *int-exp* for the corresponding formal parameter of the method; and (d) *selector-rpl* for `this`, where if *selector-exp* is a `final` local variable *var*, then *selector-rpl* is *var*, and otherwise *selector-rpl* is the owner RPL (§4.1.7) of *selector-type*.
Inferred or Missing Type Arguments, RPL Arguments, and/or *selector-exp*: If there are no explicit type or RPL arguments, the compiler infers them (if necessary) as stated in §3.4.2 and proceeds as stated above using the inferred arguments in step (4)(b). If there is no *selector-exp*, then as in ordinary Java, the implied selector is `this`.

5 Effects

An effect is a set of actions that affect memory. Every statement and expression in the program is assigned an effect. If the effects of two statements do not interfere, then the statements may be safely run in parallel.

5.1 Basic Effects

Internally, the compiler represents effects as sets of *basic effects*. A basic effect is one of the following:

1. A *read effect*, indicating a read operation on an RPL (§1) or stack region (§2.2). This effect represents any read at runtime to any heap region named by the RPL, or to the stack region.

2. A *write effect*, indicating a write operation on an RPL or stack region. This effect represents any read or write to any heap region named by the RPL, or to the stack region.

3. An *invoke effect*, indicating an invocation of method m of class C with effect E, where E is a set of basic effects.

We refer to a set of basic effects as an *effect set*.

5.2 Effect Summaries

The programmer may specify an effect summary as part of a method definition (§3.4.3). An effect summary consists either of the keyword `pure`, representing an empty effect set, or one or both of the following, in this order:

- *Read effects:* The keyword `reads`, followed by a comma-separated list of valid RPLs (§1), representing one read effect on each RPL.
- *Write effects:* The keyword `writes`, followed by a comma-separated list of valid RPLs (§1), representing one write effect on each RPL.

The effect set of the whole summary is the union of zero, one, or two effect sets generated as described above.

5.3 Effect Inference

The compiler infers the effects of a statement, expression, or statement block by using the following method-local analysis:

- For expressions *selector-exp*`.`*field-name* that directly access a non-`final` field (effects on `final` fields are ignored), the compiler computes the RPL *access-rpl* accessed by the expression. It uses the same procedure as for typing field access expressions (§4.3.1), except that it uses the the RPL specifier (§3.3) instead of the type associated with *field-name* in the class C, and there are no substitutions for type parameters. The compiler records a read or write effect (§5.1) to *access-rpl*, depending on whether the expression is used in an assignment or access.
- For statements and expressions that invoke a method, the compiler computes the effect set E generated by the invocation. It uses the same procedure as for typing method invocation expressions (§4.3.3), except that it uses the effect summary of the method (§3.4.3) instead of the return type, and there are no substitutions for type parameters. The compiler records an invocation effect (§5.1) with the method symbol and E.
- For statements and expressions that access a non-`final` stack variable (i.e., method formal parameter or local variable), record a read or write effect on a stack region identified by the variable's symbol. Effects on `final` variables are ignored.
- For any other statement or expression, the compiler accumulates the effects of its components, coarsening component effects as necessary (§5.4) to generate effects that are valid at the outer scope.

5.4 Effect Coarsening

Coarsening of RPLs: RPLs appearing in accumulated effects may include any of the following elements that are no longer in scope at the point where the effects are being reported: (1) integer variables; (2) `final` local variables of class type; and (3) locally declared region names. The compiler therefore performs the following conversion (called *effect coarsening*) to generate a valid set of accumulated effects in the surrounding environment:

- Delete all effects on RPLs starting with a local region name that is out of scope.
- Replace each RPL *rpl* that starts with *var*, where *var* is a `final` local variable not in scope, with *owner-rpl*`:*`, where *owner-rpl* is the owner RPL (§4.1.7) of the type of *var*. Perform this operation recursively on the resulting RPL.
- If an RPL *rpl* starts with `Root` or a region name that is in scope, but contains a local region name element that is out of scope, replace *rpl* with a new RPL in which the first out-of-scope name and all following elements are replaced by `*`.
- Replace all elements `[`e`]`, where e refers to variables not in scope, with `[?]`.

Coarsening of Stack Regions: If a stack region appearing in an effect goes out of scope, any effect on that stack region is deleted in translating the effect to the outer scope.

5.5 Subeffects

The following rules determine whether effect set E_1 is a subeffect of effect set E_2:

1. If E_1 and E_2 each consist of a single basic effect, then the following rules apply: (a) if R_1 is included in R_2 (§1.3.3), then a read of R_1 is a subeffect of a read or write of R_2, and a write of R_1 is a subeffect of a write of R_2; and (b) if E_3 is included in E_4, then an invocation of method M with effect E_3 is a subeffect of an invocation of the same method M with effect E_4.

2. An invocation of method M with effect E is a subeffect of E.

3. If E_1 is a subset of E_2 (i.e., all the basic effect of E_1 also appear in E_2), then E_1 is a subeffect of E_2.

4. If $E_1 = E_3 \cup E_4$, where E_3 and E_4 are subeffects of E_2, then E_1 is a subeffect of E_2.

5. The subeffect relation is reflexive and transitive.

5.6 Noninterfering Effects

The following rules determine whether effect sets E_1 and E_2 are noninterfering:

1. If E_1 and E_2 each consist of a single basic effect, then the following rules apply: (a) two read effects are noninterfering; (b) two effects on R and R', each a read or write, are noninterfering if R and R' are disjoint (§1.3.4); and (c) two invocations of the same method are noninterfering if the method is declared `commutative` (§3.4.5).

2. If E_3 and E_4 are noninterfering, then an invocation of method M with effect E_3 is noninterfering with E_4.

3. If $E_1 = E_3 \cup E_4$, where E_3 and E_4 are both noninterfering with E_2, then E_1 is noninterfering with E_2.

4. The noninterference relation is symmetric.

6 Parallel Constructs

6.1 `cobegin`

The `cobegin` statement consists of the keyword `cobegin`, followed by a statement S to execute. If S is any statement but a block enclosed in curly braces $\{\ldots\}$, or if S is a block consisting of a single statement, then the `cobegin` just executes the statement S. Otherwise, S is a block containing multiple statements, and the component statements of S are run as parallel tasks. There is an implicit barrier (join) at the end of the `cobegin` statement, so that all the component tasks must finish execution before the parent task executes the statement after the `cobegin`.

The compiler computes the inferred effect set (§5.3) of each statement in the block and checks the effect sets for pairwise noninterference (§5.6). The compiler emits a warning if interference is discovered. Both parallel and sequential code generation of `cobegin` blocks are supported.

6.2 `foreach`

The `foreach` statement has the syntax `foreach(`*index-var* `in` *start*`,` *length*`,` *stride*`)` S, where

- *index-var* is an integer variable declaration.
- *start*, *length*, and *stride* are integer expressions. The *stride* expression is optional, and the default is 1.
- S is a statement representing the loop body. It may be either a simple statement or a block.

The `foreach` statement causes one instance of S to execute for each value of *index-var* in the iteration space $\{\textit{start} + \textit{stride} \cdot i | i \in [0, \textit{length} - 1]\}$. *index-var* is treated as a `final` variable in S, so that any assignment to the index variable inside the loop causes an error.

The compiler performs the following noninterference check for indexed `foreach` loops:

1. Infer the effect set of the body of the `foreach` (§5.3).

2. Create a copy of the effect set generated in (1), but replace every occurrence of *index-var* with the negation expression (§1.3.4) of *index-var*.

If interference is detected, the compiler generates a warning.

Both parallel and sequential code generation are supported. For sequential code generation, the `foreach` iterations are executed in sequence, from lowest to highest in the iteration space. For parallel code generation, the compiler repeatedly halves the iteration space, recursively forking off pairs of parallel tasks, until a programmer-specified cutoff. The precise mechanism for specifying the cutoff is implementation dependent.

References

[1] Robert L. Bocchino and Vikram S. Adve. Formal definition and proof of soundness for Core DPJ. Technical Report UIUCDCS-R-2008-2980, University of Illinois at Urbana-Champaign, 2008.

[2] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA '09: Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 97–116, New York, NY, USA, 2009.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification (3rd Edition)*. Addison-Wesley Professional, 2005.

[4] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., New York, NY, USA, 1986.