

Dynamic Denotational Semantics of Java

Jim Alves-Foss and Fong Shing Lam

Center for Secure and Dependable Software, Department of Computer Science,
University of Idaho, Moscow ID 83844-1010, USA

Abstract. This chapter presents a dynamic denotational semantics of the Java programming language. This semantics covers almost the full range of the base language, excluding only concurrency and the API's. A discussion of these limitations is provided in the final section of the chapter.

The abstract syntax described in Chapter 1 tells us how to construct a grammatically correct program. Every syntactically correct program describes an *environment* that provides all the information about what to do during program execution. The semantics presented in this chapter, formalizes the definition of Java program behavior as defined in the *Java Language Specification (JLS)* [1]. We describe the Java environment in Section 1. Each executing program is associated with a *store* that is a repository for all instance values during program execution. The Java store is described in Section 2. Executing a Java program begins with executing the command in the static method “main” in the given class definition. Therefore, the result of a program depends on the semantics of commands and the expressions in the commands. We shall introduce a denotational semantics of these commands and expressions in Sections 3 and 4. Throughout these semantics, we concurrently define two sets of semantics, a *dynamic* and a *static* semantics, to respectively represent the execution and definitional denotations of the programs.

1 Environment

An *environment* is the information center for the execution engine and is at the heart of these semantics. Our environment is a semantic domain that has two components, the dynamic and static semantics. The *dynamic* aspect of the environment contains the traditional environmental information related to variables, their types and locations in the store (as in Stoy's classical book on denotational semantics [4]). It also contains control flow information for exceptions and breaks. The *static* aspect of the environment contains information related to all of the classes used by the program. This information includes the class members, types, initialization functions, super class and implemented interfaces. The static part of the environment is determined by evaluating the input files and then is used as an input parameter to the denotation of the main method of the invoked class.

In addition to information related to classes, their members and local variables; the environment contains a number of auxiliary variables (all starting with the symbol $\&$). These variables are used to record nesting and scoping information as well as flow control information.

- $\&package$ - specifies the fully qualified name of the package currently being defined.
- $\¤tInt$ - specifies the fully qualified name of the interface currently being defined.
- $\¤tClass$ - specifies the fully qualified name of the class currently being defined.
- $\&Mods$ - provides a list of modifiers used in the current declaration.
- $\&Type$ - defines the type used in the current declaration.
- $\&varType$ - defines if this is a “Field” or “Local” variable declaration.
- $\&switchExpr$ - value of the expression of the current Switch statement.
- $\&caseFound$ - boolean variable indicating if the case matching the switch expression has been found.
- $\&defaultFound$ - boolean variable indicating if the default switch case had been found.
- $\&caseCont$ - command continuation for execution of the appropriate switch statement. This is needed due to the fact that the default case may be defined anywhere in the switch statement.
- $\&break$ - continuation information.
- $\&return$ - specifies the command continuation to execute upon return.
- $\&returnVal, \&returnType$ - specify the return type and value for a call.
- $\&super$ - specifies the name of the current executing classes super class
- $\&throw$ - specifies the command continuation to be executed upon a throw command.
- $\&thrown$ - specifies the value, type pair referring the thrown object (exception).
- $\&thisObject$ - specifies a reference to the current object in which execution is occurring.
- $\&thisClass$ - specifies a reference to the class of the current object in which execution is occurring.

To simplify the semantic presentation, we include within the environment a collection of methods (or auxiliary functions). For these functions, we use method invocation notation for these functions, where $\gamma.m(p_1 \dots p_n)$ denotes invocation of auxiliary function m , with parameters $(p_1 \dots p_n)$, invoked in the context of the current environment, γ . (Note that variable names will be referenced in the usual way with $\gamma[name]$ referring to the current value of $name$ in the environment, and $\gamma[name \leftarrow v]$, denoting the new environment with name now returning the value v . The functions related to the dynamic semantics (execution) of a program are:

- $assignCompatible(\tau, \tau_1)$. This boolean function returns true if a value of type τ can be assigned to a variable of type τ_1 , according to the rules of the JLS [1].

- *classLoader(name, store)*. In Java, whenever a new instance of a class is created, we invoke the class loader. In the runtime system this involves first determining if the class is already loaded, otherwise finding it from some source location, loading the bytecode, and then instantiating the class constant variables and executing any static class initializer. This function represents this complex operation, and may result in modification to both the environment and the store.
- *condTypeOf(τ_t, v_t, τ_f, v_f)*. This function returns the type of conditional expressions as defined by the JLS [1]. A full definition of this function appears following the specification of condition expressions in the semantics that follow.
- *getArrayElem(a, ind)*. This function returns the location of array element *ind* from the array referenced by *a*.
- *getArrayElemType(a)*. This function returns the type of array elements in the array referenced by array *a*.
- *getArrayRef(name)*. This function returns the reference for the array *name*.
- *getComCont(term)*. This function retrieves the command condition from the environment auxiliary variable denoted by *term*, where this term can be *&break* or *&continue*. Continuations are discussed in detail in section 3.3.
- *getMethod(name, signature)*. This function returns the denotation of the named method (of the specified signature). Specifically the value returned is a semantic function that takes a set of arguments as parameters and returns a command function (a function that takes an environment, command continuation and a store and returns an answer).. All appropriate searching of the nested class and interface definitions is conducted, in accordance with the JLS [1].
- *getValue(term)*. This function returns the value of the auxiliary variable referred to by *term*.

The auxiliary functions used to build the static (declaration) portion of the environment are:

- *addConstr(mods, defn, throws, body)*. This function is used to add the constructor specification to the environment for the current class.
- *addField(name, initExp)*. This function is used to add the specified field and initialization expression to the environment given the current type and class scope.
- *addLocal(name, initExp)*. This function is used to add the specified local variable and initialization expression to the environment, given the current type and class scope.
- *addMethod(mod, hdrInfo, throws, body)*. This function is used to add the method specification to the environment for the current class.
- *addMethodHdr(hdrInfo)*. This function is used to add the abstract method header specification to the environment for the current interface.
- *addStaticField(name, valExpr)*. This function is used to add the specified static field and initialization expression to the environment given the current type and class scope.

- *addStaticInit(com)*. This function is used to add the command code (or denotation) for the specified static initializer to the class specification in the environment.
- *enterClass(mods, id, super, interfaces)*. This function is used to denote that we are currently parsing a class specification. It modifies the current scope of the environment and sets the *¤tClass* field of the environment to indicate the current class.
- *enterInterface(mods, id, extends)*. This function is used to denote that we are currently parsing an interface specification. It modifies the current scope of the environment and sets the *¤tInt* field of the environment to indicate the current interface.
- *import(name)* and *importOnDemand(name)*. These function are used to denote the java `import` command. Specifically they are used to add all the class definitions from the specified files to the environment.
- *instanceOf(τ_1, τ_2)* - returns true if τ_1 is an instance of τ_2 in the current environment.
- *isStatic()*. This boolean function returns true if the modifier of the current field declaration include the `static` modifier.

2 Store

The *store* is memory that is dynamically created, expanded, and destroyed by the execution engine. We can view the store as a communication channel between statements. Together with the environment of Section 1, it forms the *state* of the execution environment. Every local variable declaration, loading of a class object or `new` operator applied to a class type or array type creates one or more entries in the store. If the entry is a class object, the content of the entry is filled according to the constructor code of the class and field initializations. An array object is initialized with a field name of “length” denoting the number of elements in the array.

For the semantic presented in this chapter the only auxiliary function for the store is:

- *mkException(className)*. This function creates a new exception object in the store, as defined by the exception class referred to by *className*.

3 Denotational Semantics

This section presents the (almost) full denotational semantics of the Java language – only missing aspects of concurrency.

3.1 Semantics Domains and Data Values

One is often tempted, when developing a formal model of a language, to abstract out the limitations of the concrete representation of the language. For example,

authors of many language models will abstract values of type `int` to mathematical integers. Unfortunately, this provides an unrealistic definition of the behavior and meaning of the language constructs. For example, in the Java language there are no run-time indications of overflow or underflow of integers operations, but rather an implicit truncation of the resulting two's complement representation of the number to the requisite number of bits. Without an understanding of this functionality of the language and an explicit representation of it in a formal description of the language, correctness proofs of the code may be incorrect. To avoid this difficulty we represent all concrete limitations of the Java language in the following semantics of expressions. This is possible since Java precisely defines these limitations for all primitive types.

3.2 Semantic Domains

The semantic domains representing the values of the numeric data types are defined below. To simplify the semantics, we have added two special values to each of these domains, \perp (“bottom”) which represented a value with no information content and \top (“top”) which represents a value with full (potentially conflicting information). The purpose of these values is to enable each domain to be a *complete partial order*, which simplifies the mathematics underlying the semantics. These values are used by the semantic functions and do not have an equivalent representation within the Java language. The basic domains are flat domains in that there is no implicit ordering between values of the domains other than between the values and \perp and \top .

Let \mathcal{I} represent the set of integers, and \mathcal{R} represent the set of real numbers. In the following **IEEE**(*s.m.e*) denotes an IEEE 754 floating point number with sign, mantissa and exponent, **NAN** represents not-a-number and $+\infty$ and $-\infty$ represent positive and negative infinity, respectively [2].

$$\begin{aligned}
 \text{Byte} &= \{n \in \mathcal{I} \mid -128 \leq n \leq 127\} \cup \\
 &\quad \{\perp, \top\} \\
 \text{Short} &= \{n \in \mathcal{I} \mid -32768 \leq n \leq 32767\} \cup \\
 &\quad \{\perp, \top\} \\
 \text{Int} &= \{n \in \mathcal{I} \mid -2147483648 \leq n \leq 2147483647\} \cup \\
 &\quad \{\perp, \top\} \\
 \text{Long} &= \{n \in \mathcal{I} \mid -9223372036854775808 \leq n \leq 9223372036854775807\} \cup \\
 &\quad \{\perp, \top\} \\
 \text{Char} &= \{n \in \mathcal{I} \mid 0 \leq n \leq 65535\} \cup \\
 &\quad \{\perp, \top\} \\
 \text{Float} &= \{f \in \mathcal{R} \mid f = \mathbf{IEEE}(s.m.e), 0 \leq m \leq 2^{24} - 1 \wedge -149 \leq e \leq 104\} \cup \\
 &\quad \{\perp, \top, \mathbf{NAN}, +\infty, -\infty\} \\
 \text{Double} &= \{f \in \mathcal{R} \mid f = \mathbf{IEEE}(s.m.e), 0 \leq m \leq 2^{24} - 1 \wedge -149 \leq e \leq 104\} \cup \\
 &\quad \{\perp, \top, \mathbf{NAN}, +\infty, -\infty\}
 \end{aligned}$$

We define several other semantic domains for use within the denotational semantics of Java presented in this Chapter. Note that we deliberately avoid specifying the detailed semantic domains of literals, but leave them abstract and presume that a parser will interpret them correctly. Note that in this presentation here, we do not present the specifics of the domains, but rather try and define them in a context specific manner. For example we typically have $\sigma \in \Sigma$ be a store, and $\gamma \in \Gamma$ be an environment. The values (such as $r \in \mathcal{V}$) denote the basic values of the java language (shorts, ints, floats, etc.) and their types, $\tau \in \mathcal{T}$. We also refer to locations $l \in \mathcal{L}$ as indecies into the store. These are all flat domains, with a \perp and \top value as discussed above.

3.3 Continuations

Many of the semantic functions defined in the following sections utilize the concept of continuations. While evaluating a syntactic construct, we typically focus on one piece of the code. A continuation defines the semantics of the rest of that code (whether it be the rest of an expression, the rest of a command, all of a method, or the rest of a declaration). The results of continuations are either values of the specified semantic domains, such as environment or an *answer*. Since the core Java language does not interact with the outside world, we have left the concept of modifications to this world as an abstract answer domain. None of the semantics here modify that domain, such modification only occurs in the runtime libraries (Java API). We utilize the following continuations in these semantics:

- $\rho(\gamma)$ - package continuation. This continuation takes the environment parameter, γ , and returns an environment based on the declaratrions of the rest of the code. Note that this continuation is used in the highest level of package/code declarations.
- $\delta(\gamma)$ - declaration continuation. This continuation takes the environment parameter, γ , and returns an environment based on the declaratrions of the rest of the code. This continuation is used within specific declaration constructs.
- $\theta(\gamma, \sigma)$ - command continuation. This continuation takes the environment parameter, γ and store parameter, σ , and returns a an answer based on the denotation of the rest of the command. The denotation is dependent on the parameters specified, which are typically a modified store and a potentially modified environment from a command execution.
- $\kappa(r, \tau, \sigma)$ - expression continuation. This continuation takes a value, r , of type τ and a store parameter, σ , and returns a store based on the denotation of the rest of the program. The denotation is dependent on the typed data value specified, which is typically the result of an expression evaluation.
- $\alpha(v, \tau, l, \sigma)$ - location continuation. This continuation takes a value, v , of type, τ , location, l , and a store parameter, σ , and returns an answer based

on the denotation of the rest of the program. The denotation is dependent on the typed data value specified and location, which are typically the result of an expression evaluation and the location is the location of the variable referenced in the expression.

3.4 Semantic Functions

Within the denotational semantics, we make use of several semantic functions. These functions define the relationship between the code, as seen by the parser, and the actual operations of the resulting program. Since we are working with a full language specification (excluding multi-threading), we need to use a wider range of semantic functions than those found in simpler examples in the literature. The semantic functions used are divided into two categories, *operational* and *definitional*.

Operational Semantic Functions In the context of the Java programming language an operational semantic function is one that defines the relationship between the current language construct and the execution time behavior of the program. Specifically, operational semantic functions directly manipulate the store, resulting in a new store. Specifically, in the following semantics, the operational semantic functions are:

- Command functions $\mathcal{C}[\![c]\!]$. These functions define the meaning of Java commands. The meaning of any Java command is defined in terms of three parameters, the current environment γ , a command continuation θ , and the current store σ . The command continuation is a function that defines the behavior of the rest of the program in the context of an environment and a store. Therefore, the result of the $\mathcal{C}[\![c]\!]$ function is typically $\theta(\gamma_1, \sigma_1)$, where γ_1 and σ_1 are the new environment and store obtained from executing the command c , and $\theta(\gamma_1, \sigma_1)$ represents the behavior of the program given these values. This is not true when the command c results in an exception, or abnormal flow of control change (such as a **break** or **return** command.) In these cases, the result of the $\mathcal{C}[\![c]\!]$ function is based on a related continuation stored within the environment (e.g., see the semantics of the **break** and **return** commands.)
- Expression functions $\mathcal{E}[\![e]\!]$ and location functions $\mathcal{L}[\![e]\!]$. These functions define the meaning of Java expressions. We separate the location functions to denote those expressions that result in a value (called value expressions in the JLS [1]) and those that result in reference to some memory location (called variable expressions in the JLS[1]). Note that modification of the store must result in the assignment of a value to a location (either directly through an assignment statement or through a pre or post expressions – e.g., `i++`.)
 - The $\mathcal{E}[\![e]\!]$ functions are defined in terms of three parameters, the current environment γ , an expression continuation κ , and the current store σ . The expression continuation is a function that defines the behavior of

the rest of the program in the context of a value, type and a store. Therefore, the result of the $\mathcal{E}[[e]]$ function is typically $\kappa(v, \tau, \sigma_1)$, where v is the resulting value of type τ obtained from executing the expression e , and σ_1 is the resulting store. As with commands, exceptions do occur that may result in using a saved expression continuation instead of the current continuation.

- The $\mathcal{L}[[]]$ functions are also defined in terms of three parameters, the current environment γ , a location continuation α , and the current store σ . The location continuation is a function that defines the behavior of the rest of the program in the context of a value, type, location and a store. Therefore, the result of the $\mathcal{L}[[e]]$ function is typically $\alpha(v, \tau, l, \sigma_1)$, where v is the resulting value of type τ obtained from executing the expression e , which refers to a variable at location l , and σ_1 is the resulting store.

Definitional Semantic Functions A definitional semantic function is ancillary to the actual execution behavior of the program, but rather defines the context in which the execution takes place. The definitional functions used in the following semantics are:

- Goal $\mathcal{G}[[]]$ and Package functions $\mathcal{P}[[]]$. These functions define the high-level meaning of a Java source file, defined in terms of import files and class definitions. The goal function takes no parameters and returns an environment. The environment is subsequently used during execution and provides the full class definitions for the command and expression functions. The package function takes two parameters, the current environment γ and a package continuation function ρ . We use a continuation function here to be consistent with the style of semantics presents in the command and expression functions. The continuation function takes the newly modified environment and returns an environment based on the rest of the file.
- Declaration functions $\mathcal{D}[[]]$. These functions define the declarations of methods, classes, interfaces and other lower-level constructs within the package.
- Modifier functions $\mathcal{M}[[]]$. This function defines the list of modifiers for fields and methods.
- Type functions $\mathcal{T}[[]]$. These functions are used solely to determine specified data types. These types are calculated based on the current environment, provided as a parameter. The result of the type function is a string representation of the data type. Specifically we use the same string notation that Java bytecode uses to specify types [3]. Note that there are some cases where multiple types must be returned (for example the list of interfaces implemented by a class), in this case we just append the string representation of the types as the Java bytecode does for parameter lists.
- Value functions $\mathcal{V}[[]]$. These functions are a catch-all function that returns a value associated with the static input. A value function only takes the current environment as a parameter and returns a pair that consists of the value (as a basic type or string) and the type. Throughout these semantics we may need only the first or second element of this pair. We will select

these using the **fst** and **snd** operations on the result or by direct assignment $(r, \tau) = \mathcal{V}[[E]]\gamma$, where r is assigned the first value of the pair and τ is assigned the second.

3.5 Auxiliary Functions

- **mkArrayType** (τ, n) - this function takes the type specified by the first parameter and returns an array type of n dimensions.
- **mkMethodValue** (d, τ) - this function takes the definition specification of a method, d which is a pair consisting of a method name and the formal parameters, and a return type, τ , to create a value to store in the environment for searching and retrieving the methods.
- **binaryPromoteType** (τ_1, τ_2) - - returns the type resulting from a binary number promotion of the types τ_1 and τ_2 according to the rules of the JLS.
- **unaryPromoteType** (τ) - returns the type resulting from a unary numeric promotion of type τ according to the rules of the JLS.
- **promote** $(\tau, (r, \tau_1))$ - this function converts a value r , of type τ_1 to a compatible value of type τ following the numeric promotion rules of the JLS.
- **cast** $(\tau, (r, \tau_1))$ - this function converts a value r , of type τ_1 to a compatible value of type τ following the type casting rules of the JLS.
- **leftShift** $((r_1, \tau_1), (r_2, \tau_2))$ - this function returns the value of r_1 , of type τ_1 left shifted r_2 places (where r_2 is of type τ_2). The resulting value is of type τ_1 .
- **String** (r, τ) - this function actually invokes the `toString` function of the `java.lang` class corresponding to the type τ on the value r to return a string.
- **fst** (p) - this is a function that takes returns the first value of a pair.
- **snd** (p) - this is a function that takes returns the second value of a pair.
- **append** $(l1, l1)$ - this is a simple list append function.
- **isNumeric** (τ) - this function returns true is the type of the parameter can be classified as a numeric value.

4 Java Semantics

The following sections detail the semantics of the Java language. To simplify the presentation, we have taken the full syntax of the Java language (as presented in Chapter 2 of this volume) and reduced it, by removing high-level redundancy. For example, syntactically we define several levels of statements, including those with and without trailing if's. For the presentation here, we only worry about the actual statements, such as the for-statement, while-statement, etc.

4.1 The Meaning of a Java Program

When a user wants to execute a Java program *myclass* they type “`javac myclass args`”. In the context of the semantics presented here, this is defined as:

$(\gamma.getMethod("myclass.main", "[Ljava.lang.String;"))\mathcal{V}[\text{args}]\gamma c_{\text{exit}}\sigma$ where
 $\gamma = \mathcal{G}[\text{Goal}]$ // where Goal is the myclass.java file and
 $\sigma = \gamma.classLoader(myclass, \text{newStore}())$ and
 c_{exit} = the command continuation function that terminates the program

This semantics evaluates the source file (and all other imported class files) to create a new environment γ . It then invokes a `ClassLoader` function to load the specified class into the store, σ and executes the method `main` of the class with respect to the specified command-line arguments, the environment and the new store.

4.2 Names and Literals

The designers of the Java language separated the concept of names from other primary entities in the grammar. The reason for this is to avoid some possible ambiguities in a LALR(1) parser. For the semantic functions, all we need to return is a representation of the name to be used once the full name is defined. Since the value semantic function requires a returned (value,type) pair, we specify the type of names as "name".

$$\begin{aligned} \mathcal{V}[\langle \text{Name} \rangle]\gamma ::= & \\ & \mathcal{V}[\langle \text{SimpleName} \rangle]\gamma \\ & | \mathcal{V}[\langle \text{QualifiedName} \rangle]\gamma \\ \\ \mathcal{V}[\langle \text{SimpleName} \rangle]\gamma ::= & \\ & \mathcal{V}[\text{Id}]\gamma = (\text{ValueOf}(\text{Id}), \text{"name"}) \\ \\ \mathcal{V}[\langle \text{QualifiedName} \rangle]\gamma ::= & \\ & \mathcal{V}[\text{Name.Id}]\gamma = \\ & (\text{Str} + \text{'.'} + \text{ValueOf}(\text{Id}), \text{"name"}) \text{ where} \\ & \text{Str} = \text{fst}(\mathcal{V}[\text{Name}]\gamma) \\ \\ \mathcal{V}[\langle \text{Literal} \rangle]\gamma ::= & \\ & \mathcal{V}[\text{IntLit}]\gamma = (\text{ValueOf}(\text{IntLit}), \text{"I"}) \\ & | \mathcal{V}[\text{FloatLit}]\gamma = (\text{ValueOf}(\text{FloatLit}), \text{"F"}) \\ & | \mathcal{V}[\text{BoolLit}]\gamma = (\text{ValueOf}(\text{BoolLit}), \text{"Z"}) \\ & | \mathcal{V}[\text{CharLit}]\gamma = (\text{ValueOf}(\text{CharLit}), \text{"C"}) \\ & | \mathcal{V}[\text{StringLit}]\gamma = (\text{ValueOf}(\text{StringLit}), \text{"Ljava.lang.String;"}) \\ & | \mathcal{V}[\text{NullLit}]\gamma = (\text{null}, \text{"L;"}) \end{aligned}$$

4.3 Packages

Goal and Compilation Unit. The following productions define the semantics of a single Java compilation unit. This is encapsulated within a goal, which has no parameters. The goal semantics specify the creation of a new environment and an identity continuation such that the result of the goal will be an environment to be used during execution. In the $\langle \text{CompUnit} \rangle$ specification we forced the

automatic loading of the `java.lang` package as if there were the statement “`import java.lang.*`” immediately following any package declaration statement.

$$\begin{aligned}
\mathcal{G}[\langle \text{Goal} \rangle] &::= \\
&\mathcal{G}[\langle \text{CompUnit} \rangle] = \mathcal{P}[\langle \text{CompUnit} \rangle]\gamma\rho \text{ where} \\
&\quad \gamma = \text{newEnvironment}() \text{ and} \\
&\quad \forall \gamma_1. \rho(\gamma_1) = \gamma_1 \\
\mathcal{P}[\langle \text{CompUnit} \rangle]\gamma\rho &::= \\
&\mathcal{P}[\langle \text{PackageDecl} \rangle^? \langle \text{ImportDeclList} \rangle^? \langle \text{TypeDeclList} \rangle^?]\gamma\rho = \\
&\mathcal{P}[\langle \text{PackageDecl} \rangle^?]\gamma\rho_1 \text{ where} \\
&\quad \forall \gamma_1. \rho_1(\gamma_1) = \mathcal{P}[\langle \text{ImportDeclList} \rangle^?]\gamma_2\rho_2 \text{ where} \\
&\quad \quad \gamma_2 = \rho(\gamma.\text{importOnDemand}(\text{java.lang})) \text{ and} \\
&\quad \quad \forall \gamma_2. \rho_2(\gamma_2) = \mathcal{P}[\langle \text{TypeDeclList} \rangle^?]\gamma_2\rho_1' \\
\mathcal{P}[\langle \text{PackageDecl} \rangle]\gamma\delta &::= \\
&\mathcal{P}[\mathbf{package} \langle \text{Name} \rangle ;]\gamma\rho = \rho(\gamma\{\&package \leftarrow (\text{fst}(\mathcal{V}[\langle \text{Name} \rangle]\gamma))\})
\end{aligned}$$

Import commands. The import commands cause some difficulty in the semantics. Specifically, an import command loads into the environment all the relevant information related to an entity specified in a separate compilation unit. For the sake of brevity we include auxiliary functions that modify the environment to reflect the action of the import command.

$$\begin{aligned}
\mathcal{P}[\langle \text{ImportDeclList} \rangle]\gamma\delta &::= \\
&\mathcal{P}[\langle \text{ImportDecl} \rangle]\gamma\rho \\
&| \mathcal{P}[\langle \text{ImportDeclList}_1 \rangle \langle \text{ImportDecl} \rangle]\gamma\rho = \\
&\mathcal{P}[\langle \text{ImportDeclList}_1 \rangle]\gamma\rho_1 \text{ where} \\
&\quad \forall \gamma_1. \rho_1(\gamma_1) = \mathcal{P}[\langle \text{ImportDecl} \rangle]\gamma_1\rho \\
\mathcal{P}[\langle \text{ImportDecl} \rangle]\gamma\rho &::= \\
&\mathcal{P}[\langle \text{SingleTypeImportDecl} \rangle]\gamma\rho \\
&| \mathcal{P}[\langle \text{TypeImportOnDemandDecl} \rangle]\gamma\rho \\
\mathcal{P}[\langle \text{SingleTypeImportDecl} \rangle]\gamma\delta &::= \\
&\mathcal{P}[\mathbf{import} \langle \text{Name} \rangle ;]\gamma\rho = \rho(\gamma.\text{import}(\text{fst}(\mathcal{V}[\langle \text{Name} \rangle]\gamma))) \\
\mathcal{P}[\langle \text{TypeImportOnDemandDecl} \rangle]\gamma\delta &::= \\
&\mathcal{P}[\mathbf{import} \langle \text{Name} \rangle . * ;]\gamma\rho = \rho(\gamma.\text{importOnDemand}(\text{fst}(\mathcal{V}[\langle \text{Name} \rangle]\gamma)))
\end{aligned}$$

Class and Interface Declarations. The class and interface declaration productions are defined in terms of the declaration semantic function. The following semantics define the relationship between the package and declaration semantic functions.

$$\begin{aligned}
\mathcal{P}[\langle \text{TypeDeclList} \rangle]\gamma\delta &::= \\
&\mathcal{P}[\langle \text{TypeDecl} \rangle]\gamma\rho \\
&| \mathcal{P}[\langle \text{TypeDeclList} \rangle \langle \text{TypeDecl} \rangle]\gamma\rho = \\
&\mathcal{P}[\langle \text{TypeDeclList} \rangle]\gamma\rho_1 \text{ where}
\end{aligned}$$

$$\begin{aligned} \forall \gamma_1. \rho_1(\gamma_1) &= \mathcal{D}[\langle \text{TypeDecl} \rangle] \gamma_1 \delta \text{ where} \\ \forall \gamma_1. \delta(\gamma_1) &= \rho(\gamma_1) \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{TypeDecl} \rangle] \gamma \delta &::= \\ &\mathcal{D}[\langle \text{ClassDecl} \rangle] \gamma \delta \\ &| \mathcal{D}[\langle \text{InterfaceDecl} \rangle] \gamma \delta \\ &| \mathcal{D}[\langle \text{;} \rangle] \gamma \delta = \delta(\gamma) \end{aligned}$$

4.4 Types

The $\mathcal{T}[\]$ semantic function returns the type defined by the given syntactic construct. The returned type is a string representation of the specified type using the notation defined in the JVM [3]. Specifically, the returned values are:

Type	Return Value
boolean	“Z”
byte	“B”
short	“S”
char	“C”
int	“I”
long	“J”
float	“F”
double	“D”
void	“V”
array of <i>Type</i>	“[<i>Type</i> ”
class or interface	“L <i>classname</i> ,”
method*	(<i>t</i> ₁ <i>t</i> ₂ ... <i>t</i> _{<i>n</i>}) <i>t</i> _{<i>r</i>}

for methods of the form:
*return-type meth-name(parm*₁*, parm*₂*, ..., parm*_{*n*}*)*
 where *t*_{*r*} is the return type, *t*_{*i*} is the type of *parm*_{*i*}.

$$\begin{aligned} \mathcal{T}[\langle \text{Type} \rangle] \gamma &::= \\ &\mathcal{T}[\langle \text{PrimitiveType} \rangle] \gamma \\ &| \mathcal{T}[\langle \text{ReferenceType} \rangle] \gamma \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{PrimitiveType} \rangle] \gamma &::= \\ &\mathcal{T}[\langle \text{NumericType} \rangle] \gamma \\ &| \mathcal{T}[\mathbf{boolean}] \gamma = \text{“Z”} \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{NumericType} \rangle] \gamma &::= \\ &\mathcal{T}[\langle \text{IntegralType} \rangle] \gamma \\ &\mathcal{T}[\langle \text{FloatingPointType} \rangle] \gamma \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{IntegralType} \rangle] \gamma &::= \\ &\mathcal{T}[\mathbf{byte}] \gamma = \text{“B”} \\ &| \mathcal{T}[\mathbf{int}] \gamma = \text{“I”} \\ &| \mathcal{T}[\mathbf{long}] \gamma = \text{“J”} \\ &| \mathcal{T}[\mathbf{short}] \gamma = \text{“S”} \\ &| \mathcal{T}[\mathbf{char}] \gamma = \text{“C”} \end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\langle \text{FloatingPointType} \rangle] \gamma &::= \\
&\quad \mathcal{T}[\mathbf{float}] \gamma = \text{"F"} \\
&\quad | \mathcal{T}[\mathbf{double}] \gamma = \text{"D"} \\
\mathcal{T}[\langle \text{ReferenceType} \rangle] \gamma &::= \\
&\quad \mathcal{T}[\langle \text{ClassOrInterfaceType} \rangle] \gamma \\
&\quad | \mathcal{T}[\langle \text{ArrayType} \rangle] \gamma \\
\mathcal{T}[\langle \text{ClassOrInterfaceType} \rangle] \gamma &::= \\
&\quad \mathcal{T}[\langle \text{name} \rangle] \gamma = \text{"L"} + \text{fst}(\mathcal{V}[\langle \text{Name} \rangle] \gamma) + \text{";"} \\
\mathcal{T}[\langle \text{ClassType} \rangle] \gamma &::= \\
&\quad \mathcal{T}[\langle \text{ClassOrInterfaceType} \rangle] \gamma \\
\mathcal{T}[\langle \text{InterfaceType} \rangle] \gamma &::= \\
&\quad \mathcal{T}[\langle \text{ClassOrInterfaceType} \rangle] \gamma \\
\mathcal{T}[\langle \text{ArrayType} \rangle] \gamma &::= \\
&\quad \mathcal{T}[\langle \text{PrimitiveType} \rangle [\]] \gamma = \mathbf{mkArrayType}(\tau_1, 1) \text{ where} \\
&\quad \quad \tau_1 = \mathcal{T}[\langle \text{PrimitiveType} \rangle] \gamma \\
&\quad | \mathcal{T}[\langle \text{Name} \rangle [\]] \gamma = \mathbf{mkArrayType}(\tau_1, 1) \text{ where} \\
&\quad \quad \tau_1 = \mathbf{fst}(\mathcal{V}[\langle \text{Name} \rangle] \gamma) \\
&\quad | \mathcal{T}[\langle \text{ArrayType} \rangle [\]] \gamma = \mathbf{mkArrayType}(\tau_1, 1) \text{ where} \\
&\quad \quad \tau_1 = \mathcal{T}[\langle \text{ArrayType} \rangle] \gamma \text{ where}
\end{aligned}$$

$$\mathbf{mkArrayType}(\tau, n) = \begin{cases} \tau & \text{when } n = 0 \\ \text{"["} + \mathbf{mkArrayType}(\tau, n - 1) & \text{when } n > 0 \end{cases}$$

4.5 Modifiers

Modifiers specify the access constraints of classes, methods and fields in Java programs. As such, we need to specify the list of modifiers for the declaration semantic functions that use them. The $\mathcal{M}[\]$ semantic function returns all modifiers as a list of strings.

$$\begin{aligned}
\mathcal{M}[\langle \text{Modifiers} \rangle] &::= \\
&\quad \mathcal{M}[\langle \text{Modifier} \rangle] \\
&\quad | \mathcal{M}[\langle \text{Modifiers}_1 \rangle \langle \text{Modifier} \rangle] = \text{cons}(\mathcal{M}[\langle \text{Modifiers}_1 \rangle], \mathcal{M}[\langle \text{Modifier} \rangle]) \\
\mathcal{M}[\langle \text{Modifier} \rangle] &::= \\
&\quad \mathcal{M}[\mathbf{public}] = \text{"public"} \\
&\quad | \mathcal{M}[\mathbf{private}] = \text{"private"} \\
&\quad | \mathcal{M}[\mathbf{protected}] = \text{"protected"} \\
&\quad | \mathcal{M}[\mathbf{static}] = \text{"static"} \\
&\quad | \mathcal{M}[\mathbf{abstract}] = \text{"abstract"} \\
&\quad | \mathcal{M}[\mathbf{final}] = \text{"final"} \\
&\quad | \mathcal{M}[\mathbf{native}] = \text{"native"}
\end{aligned}$$

| $\mathcal{M}[\mathbf{synchronized}] = \text{“synchronized”}$
 | $\mathcal{M}[\mathbf{transient}] = \text{“transient”}$
 | $\mathcal{M}[\mathbf{volatile}] = \text{“volatile”}$

4.6 Interface Declarations

Interfaces specify a group of classes. Within each interface is a set of nested class and interface declarations (starting in Java 1.1), constant fields and abstract methods. In addition, an interface can extend another interface. All of this syntax represents a declaration abstraction for a group of classes. When a class is declared to implement an interface, all of the interface body declarations are included in the beginning of the class declaration. The auxiliary function *enterInterface* modifies the current environment to include the new interface declaration and members. The current interface is defined in terms of the name of the declared interface for the remainder of the declaration, it then reverts to the calling interface name for the continuation.

$$\begin{aligned} \mathcal{D}[\langle \text{InterfaceDecl} \rangle] \gamma \delta ::= & \\ \mathcal{D}[\langle \text{Modifiers} \rangle^? \mathbf{interface} \langle \text{Id} \rangle \langle \text{Extends} \rangle^? \langle \text{InterfaceBody} \rangle] \gamma \delta = & \\ \mathcal{D}[\langle \text{InterfaceBody} \rangle] \gamma_1 \delta_1 \text{ where} & \\ \gamma_1 = \gamma.\mathit{enterInterface}(\mathcal{M}[\langle \text{Modifiers} \rangle], & \\ \mathcal{V}[\langle \text{Id} \rangle] \gamma, & \\ \mathcal{T}[\langle \text{Extends} \rangle] \gamma) \text{ and} & \\ \forall \gamma_2. \delta_1(\gamma_2) = \delta(\gamma_2[\&\mathit{currentInt} \leftarrow \gamma(\&\mathit{currentInt})]) & \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{Extends} \rangle] \gamma ::= & \\ \mathcal{T}[\mathbf{extends} \langle \text{InterfaceType} \rangle] \gamma = \mathcal{T}[\langle \text{InterfaceType} \rangle] \gamma & \\ | \mathcal{T}[\langle \text{Extends} \rangle, \langle \text{InterfaceType} \rangle] \gamma = & \\ \mathcal{T}[\langle \text{Extends} \rangle] \gamma + \mathcal{T}[\langle \text{InterfaceType} \rangle] \gamma & \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{InterfaceBody} \rangle] \gamma \delta ::= & \\ \mathcal{D}[\{ \langle \text{InterfaceMemberDeclList} \rangle^? \}] \gamma \delta = & \\ \mathcal{D}[\langle \text{InterfaceMemberDeclList} \rangle] \gamma \delta & \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{InterfaceMemberDeclList} \rangle] \gamma \delta ::= & \\ \mathcal{D}[\langle \text{InterfaceMemberDecl} \rangle] \gamma \delta & \\ | \mathcal{D}[\langle \text{InterfaceMemberDeclList}_1 \rangle \langle \text{InterfaceMemberDecl} \rangle] \gamma \delta = & \\ \mathcal{D}[\langle \text{InterfaceMemberDecl} \rangle] \gamma \delta_1 \text{ where} & \\ \forall \gamma_1. \delta_1(\gamma_1) = \mathcal{D}[\langle \text{InterfaceMemberDecl} \rangle] \gamma_1 \delta & \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{InterfaceMemberDecl} \rangle] \gamma \delta ::= & \\ \mathcal{D}[\langle \text{ClassDecl} \rangle] \gamma \delta & \\ | \mathcal{D}[\langle \text{InterfaceDecl} \rangle] \gamma \delta & \\ | \mathcal{D}[\langle \text{AbsMethodDecl} \rangle] \gamma \delta & \\ | \mathcal{D}[\langle \text{ConstantDecl} \rangle] \gamma \delta & \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{AbsMethodDecl} \rangle] \gamma \delta ::= & \\ \mathcal{D}[\langle \text{MethodHdr} \rangle ;] \gamma \delta = \delta(\gamma_1) \text{ where} & \\ \gamma_1 = \gamma.\mathit{addMethodHdr}(v) \text{ and} & \end{aligned}$$

$$v = \mathcal{V}[\langle \text{MethodHdr} \rangle] \gamma$$

$$\begin{aligned} \mathcal{D}[\langle \text{ConstantDecl} \rangle] \gamma \delta &::= \\ \mathcal{D}[\langle \text{FieldDecl} \rangle] \gamma \delta \end{aligned}$$

4.7 Class Declarations

The following grammar presents class declarations. As with interfaces, a class declaration enters a new class, thus modifying the environment. The environment includes the definitions of all members of the class and links to the super class and implemented interfaces.

$$\begin{aligned} \mathcal{D}[\langle \text{ClassDecl} \rangle] \gamma \delta &::= \\ \mathcal{D}[\langle \text{Modifiers} \rangle? \text{ class } \langle \text{Id} \rangle \langle \text{Super} \rangle? \langle \text{Interfaces} \rangle? \langle \text{ClassBody} \rangle] \gamma \delta &= \\ \mathcal{D}[\langle \text{ClassBody} \rangle] \gamma_1 \delta_1 \text{ where} & \\ \gamma_1 = \gamma.\text{enterClass}(\mathcal{M}[\langle \text{Modifiers} \rangle]), & \\ \mathcal{V}[\langle \text{Id} \rangle] \gamma, & \\ \mathcal{T}[\langle \text{Super} \rangle] \gamma, & \\ \mathcal{T}[\langle \text{Interfaces} \rangle] \gamma \text{ and} & \\ \forall \gamma_2. \delta_1(\gamma_2) = \delta(\gamma_2[\¤tClass \leftarrow \gamma(\¤tClass)]) & \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{Super} \rangle] \gamma &::= \\ \mathcal{T}[\text{extends } \langle \text{ClassType} \rangle] \gamma = \mathcal{T}[\langle \text{ClassType} \rangle] \gamma \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{Interfaces} \rangle] \gamma &::= \\ \mathcal{T}[\text{implements } \langle \text{InterfaceTypeList} \rangle] \gamma = \mathcal{T}[\langle \text{InterfaceTypeList} \rangle] \gamma \end{aligned}$$

$$\begin{aligned} \mathcal{T}[\langle \text{InterfaceTypeList} \rangle] \gamma &::= \\ \mathcal{T}[\langle \text{InterfaceType} \rangle] \gamma & \\ | \mathcal{T}[\langle \text{InterfaceTypeList}_1 \rangle, \langle \text{InterfaceType} \rangle] \gamma = & \\ \mathcal{T}[\langle \text{InterfaceTypeList}_1 \rangle] \gamma + \mathcal{T}[\langle \text{InterfaceType} \rangle] \gamma & \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{ClassBody} \rangle] \gamma \delta &::= \\ \mathcal{D}[\langle \text{ClassBodyDeclList}^? \rangle] \gamma \delta \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{ClassBodyDeclList} \rangle] \gamma \delta &::= \\ \mathcal{D}[\langle \text{ClassBodyDecl} \rangle] \gamma \delta & \\ | \mathcal{D}[\langle \text{ClassBodyDeclList}_1 \rangle \langle \text{ClassBodyDecl} \rangle] \gamma \delta \end{aligned}$$

The Class Body consists of class members which include nested classes, nested interfaces, fields and methods; constructors and static initializers (which are class-level constructors invoked the first time a class is activated). It is important to note that when a class is activated (denoted in these semantics by the *ClassLoader* function): the parent class is loaded, then all static variables are initialized and all static initializers are executed in the order they appear in the class declaration. The *addStaticInit* routine used below, and the *addStaticField* routine used in the Field Declaration section enters the partial semantic functions for these initializers into the environment. The *ClassLoader* function recovers these partial functions and completes them with the current parameters.

$$\begin{aligned} \mathcal{D}[\langle \text{ClassBodyDecl} \rangle] \gamma \delta ::= & \\ & \mathcal{D}[\langle \text{ClassMemberDecl} \rangle] \gamma \delta \\ & | \mathcal{D}[\langle \text{StaticInit} \rangle] \gamma \delta = \\ & \quad \delta(\gamma.\text{addStaticInit}(\mathcal{C}[\langle \text{StaticInit} \rangle])) \\ & | \mathcal{D}[\langle \text{ConstrDecl} \rangle] \gamma \delta \end{aligned}$$

$$\begin{aligned} \mathcal{D}[\langle \text{ClassMemberDecl} \rangle] \gamma \delta ::= & \\ & \mathcal{D}[\langle \text{ClassDecl} \rangle] \gamma \delta \\ & | \mathcal{D}[\langle \text{InterfaceDecl} \rangle] \gamma \delta \\ & | \mathcal{D}[\langle \text{FieldDecl} \rangle] \gamma \delta \\ & | \mathcal{D}[\langle \text{MethodDecl} \rangle] \gamma \delta \end{aligned}$$

4.8 Method Declarations

We have slightly modified the method declaration productions from the grammar presented in the JLS [1], including modifiers and throws directly in the method declaration instead of in the header. This was done to simplify the semantic functions. The major action of these productions is to add a method into the environment of the current class. Associated with the type signature and name of the method is a partial semantic function that defines the operational behavior of the method. When the method is invoked, values of the actual parameters are passed to the formal parameters of the method, and then the method body is executed using the new values. Any resultant value is returned as the result of the method semantic function. The **mkMethodValue** function takes the method name and formal parameter type list and returns what we term a method value. This method value specifies the name and type signature of the method along with the names of the formal parameters. The exact details of this notation is not important here, it is sufficient to know that this information is used by the *addMethod* routine.

$$\begin{aligned} \mathcal{D}[\langle \text{MethodDecl} \rangle] \gamma \delta ::= & \\ & \mathcal{D}[\langle \text{Modifiers} \rangle^? \langle \text{MethodHdr} \rangle \langle \text{Throws} \rangle^? \langle \text{MethodBody} \rangle] \gamma \delta = \\ & \quad \gamma.\text{addMethod}(\mathcal{M}[\langle \text{Modifiers} \rangle], \mathcal{V}[\langle \text{MethodHdr} \rangle] \gamma, \\ & \quad \quad \mathcal{T}[\langle \text{Throws} \rangle] \gamma, \mathcal{C}[\langle \text{MethodBody} \rangle]) \end{aligned}$$

$$\begin{aligned} \mathcal{V}[\langle \text{MethodHdr} \rangle] \gamma ::= & \\ & \mathcal{V}[\langle \text{Type} \rangle \langle \text{MethodDef} \rangle] \gamma = \mathbf{mkMethodValue}(\mathcal{V}[\langle \text{MethodDef} \rangle] \gamma, \langle \text{Type} \rangle) \\ & | \mathcal{V}[\mathbf{void} \langle \text{MethodDef} \rangle] \gamma = \mathbf{mkMethodValue}(\mathcal{V}[\langle \text{MethodDef} \rangle] \gamma, \mathbf{void}) \end{aligned}$$

$$\begin{aligned} \mathcal{V}[\langle \text{MethodDef} \rangle] \gamma ::= & \\ & \mathcal{V}[\langle \text{Id} \rangle (\langle \text{FormalParmList} \rangle^?)] \gamma = \\ & \quad (\text{fst}(\mathcal{V}[\langle \text{Id} \rangle] \text{env}), \mathcal{V}[\langle \text{FormalParmList} \rangle] \gamma) \\ & | \mathcal{V}[\langle \text{MethodDef} \rangle [\]] \gamma = \mathbf{mkArrayType}(\mathcal{V}[\langle \text{MethodDef} \rangle] \gamma, 1) \end{aligned}$$

The formal parameter list is returns a pair that consists of a list names of each of the parameters and a list of corresponding types. The following semantic functions return this pair. The throws clause returns a type that corresponds to the types of each of the thrown classes.

$$\begin{aligned}
\mathcal{V}[\langle \text{FormalParamList} \rangle] \gamma &::= \\
&\mathcal{V}[\langle \text{FormalParam} \rangle] \gamma \\
&| \mathcal{V}[\langle \text{FormalParamList} \rangle, \langle \text{FormalParam} \rangle] \gamma = \\
&\quad \mathcal{V}[\langle \text{FormalParamList} \rangle] \gamma + \mathcal{V}[\langle \text{FormalParam} \rangle] \gamma \\
\mathcal{V}[\langle \text{FormalParam} \rangle] \gamma &::= \\
&\mathcal{T}[\langle \text{Modifier} \rangle \langle \text{Type} \rangle \langle \text{VarDeclId} \rangle] \gamma = (\text{fst}(\mathcal{V}[\langle \text{VarDeclId} \rangle]), \mathcal{T}[\langle \text{Type} \rangle] \gamma) \\
\mathcal{T}[\langle \text{Throws} \rangle] \gamma &::= \\
&\mathcal{T}[\mathbf{throws} \langle \text{ClassTypeList} \rangle] \gamma = \mathcal{T}[\langle \text{ClassTypeList} \rangle] \gamma \\
\mathcal{T}[\langle \text{ClassTypeList} \rangle] \gamma &::= \\
&\mathcal{T}[\langle \text{ClassType} \rangle] \gamma \\
&| \mathcal{T}[\langle \text{ClassTypeList} \rangle, \langle \text{ClassType} \rangle] \gamma \delta = \\
&\quad \mathcal{T}[\langle \text{ClassTypeList} \rangle] \gamma + \mathcal{T}[\langle \text{ClassType} \rangle] \gamma \\
\mathcal{C}[\langle \text{MethodBody} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{C}[\mathbf{;}] \gamma \theta \sigma = \theta(\gamma, \sigma) \\
&| \mathcal{C}[\langle \text{Block} \rangle] \gamma \delta \sigma
\end{aligned}$$

4.9 Field and Variable Declarations

The following semantic functions define the meaning of the field and variable declarations. These declarations are used to modify the environment to define static and regular fields and local variables.

$$\begin{aligned}
\mathcal{D}[\langle \text{FieldDecl} \rangle] \gamma \delta &::= \\
&\mathcal{D}[\langle \text{Modifiers} \rangle^? \langle \text{Type} \rangle \langle \text{VarDecl} \rangle \mathbf{;}] \gamma \delta = \mathcal{D}[\langle \text{VarDecl} \rangle] \gamma_1 \delta \text{ where} \\
&\quad \gamma_1 = \gamma[\&Mods \leftarrow \mathcal{M}[\langle \text{Modifiers} \rangle], \\
&\quad \quad \&Type \leftarrow \mathcal{T}[\langle \text{Type} \rangle] \gamma, \\
&\quad \quad \&varType \leftarrow \text{"Field"}] \\
\mathcal{D}[\langle \text{VarDeclList} \rangle] \gamma \delta &::= \\
&\mathcal{D}[\langle \text{VarDecl} \rangle] \gamma \delta \\
&| \mathcal{D}[\langle \text{VarDeclList} \rangle, \langle \text{VarDecl} \rangle] \gamma \delta = \mathcal{D}[\langle \text{VarDeclList} \rangle] \gamma \delta_1 \text{ where} \\
&\quad \forall \gamma_1. \delta_1(\gamma_1) = \mathcal{D}[\langle \text{VarDecl} \rangle] \gamma_1 \delta \\
\mathcal{D}[\langle \text{VarDecl} \rangle] \gamma \delta &::= \\
&\mathcal{D}[\langle \text{VarDeclId} \rangle] \gamma \delta = \delta(\gamma_1) \text{ where} \\
&\quad \text{let } (name, type) = \mathcal{V}[\langle \text{VarDeclId} \rangle] \gamma \text{ in} \\
&\quad \gamma_1 = \\
&\quad \quad \text{if } (\gamma(\&varType) == \text{"Field"}) \\
&\quad \quad \quad \text{if } (\gamma.isStatic()) \\
&\quad \quad \quad \quad \gamma.addStaticField(name, defaultInit(type)) \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \gamma.addField(name, defaultInit(type)) \\
&\quad \quad \quad \text{endif} \\
&\quad \quad \text{else } //(\gamma(\&varType) == \text{"Local"}) \\
&\quad \quad \quad \gamma.addLocal(name, defaultInit(type))
\end{aligned}$$

```

    endif
  |  $\mathcal{D}[\langle \text{VarDeclId} \rangle] \gamma \delta = \langle \text{VarInit} \rangle \gamma \delta = \delta(\gamma_1)$  where
    let  $(name, type) = \mathcal{V}[\langle \text{VarDeclId} \rangle] \gamma$  in
       $\gamma_1 =$ 
        if  $(\gamma(\&varType) == \text{"Field"})$ 
          if  $(\gamma.isStatic())$ 
             $\gamma.addStaticField(name, \mathcal{E}[\langle \text{VarInit} \rangle])$ 
          else
             $\gamma.addField(name, \mathcal{E}[\langle \text{VarInit} \rangle])$ 
          endif
        else  $//(\gamma(\&varType) == \text{"Local"})$ 
           $\gamma.addLocal(name, \mathcal{E}[\langle \text{VarInit} \rangle])$ 
        endif

 $\mathcal{V}[\langle \text{VarDeclId} \rangle] \gamma ::=$ 
   $\mathcal{V}[\langle \text{Id} \rangle] \gamma = (\text{ValueOf}(\langle \text{Id} \rangle), \gamma(\&Type))$ 
  |  $\mathcal{D}[\langle \text{VarDeclId} \rangle [ ]] \gamma \delta = (\text{ValueOf}(\langle \text{Id} \rangle), \mathbf{mkArrayType}(\gamma(\&Type), 1))$ 

```

4.10 Initializers

Initializers consist of both static block initializers for classes and field and local variable initializers. All initializers are simply evaluated upon instantiation of the class, field or variable. For fields and variables the resultant value is a pair consisting of a list of values and a list of types corresponding to these values.

```

 $\mathcal{C}[\langle \text{StaticInit} \rangle] \gamma \theta \sigma ::=$ 
   $\mathcal{C}[\mathbf{static} \langle \text{Block} \rangle] \gamma \theta \sigma = \mathcal{C}[\langle \text{Block} \rangle] \gamma \theta \sigma$ 

 $\mathcal{E}[\langle \text{VarInits} \rangle] \gamma \kappa \sigma ::=$ 
   $\mathcal{E}[\langle \text{VarInit} \rangle] \gamma \kappa \sigma$ 
  |  $\mathcal{E}[\langle \text{VarInits}_1 \rangle, \langle \text{VarInit} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{VarInits}_1 \rangle] \gamma \kappa_1 \sigma$  where
     $\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{VarInit} \rangle] \gamma \kappa_2 \sigma_1$  where
       $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$  where
         $q = \mathbf{append}(r_1, r_2)$  and
         $\tau = \tau_1 + \tau_2$ 

 $\mathcal{E}[\langle \text{VarInit} \rangle] \gamma \kappa \sigma ::=$ 
   $\mathcal{E}[\langle \text{Expression} \rangle] \gamma \kappa \sigma$ 
   $\mathcal{E}[\langle \text{ArrayInitializer} \rangle] \gamma \kappa \sigma$ 

 $\mathcal{E}[\langle \text{ArrayInit} \rangle] \gamma \kappa \sigma ::=$ 
   $\mathcal{E}[\{ \langle \text{VarInits} \rangle^?, ? \}] \gamma \kappa \sigma = \mathcal{E}[\langle \text{VarInits} \rangle] \gamma \kappa_1 \sigma$  where
     $\forall r, \tau, \sigma. \kappa_1(r, \tau, \sigma) = \kappa(r, \tau_1, \sigma)$  where
       $\tau_1 = \mathbf{mkArrayType}(\tau, 1)$ 

```

4.11 Constructor Declarations

The constructor semantic functions define the meaning of object constructors. It is important to understand that when a constructor is invoked, it either calls an

implicit constructor of the super class or an explicit constructor. This is denoted in the semantic functions below. The *instantiateClass* function of stores return a triple consisting of a value (the reference to the new object), a type (of the object), and a new store that contains the new locations for the fields of the object.

$$\begin{aligned} \mathcal{D}[\langle \text{ConstrDecl} \rangle] \gamma \delta ::= & \\ & \mathcal{D}[\langle \text{Modifiers} \rangle^? \langle \text{ConstrDef} \rangle \langle \text{Throws} \rangle^? \langle \text{ConstrBody} \rangle] \gamma \delta = \delta(\gamma_1) \text{ where} \\ & \gamma_1 = \gamma.addConstr(\mathcal{M}[\langle \text{Modifiers} \rangle], \mathcal{V}[\langle \text{ConstrDef} \rangle] \gamma, \\ & \quad \mathcal{T}[\langle \text{Throws} \rangle] \gamma, \mathcal{E}[\langle \text{ConstrBody} \rangle]) \end{aligned}$$

$$\begin{aligned} \mathcal{V}[\langle \text{ConstrDef} \rangle] \gamma ::= & \\ & \mathcal{V}[\langle \text{SimpleName} \rangle (\langle \text{FormalParmList} \rangle^?)] \gamma = \\ & (\text{fst}(\mathcal{V}[\langle \text{SimpleName} \rangle] \gamma), \mathcal{V}[\langle \text{FormalParmList} \rangle] \gamma) \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{ConstrBody} \rangle] \gamma \kappa \sigma ::= & \\ & \mathcal{C}[\{ \langle \text{ExplConstrInv} \rangle \langle \text{BlockStmtList} \rangle^? \}] \gamma \kappa \sigma = \\ & \mathcal{E}[\langle \text{ExplConstrInv} \rangle] \gamma \kappa_1 \sigma \text{ where} \\ & \quad \forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) = \theta(\sigma_1) \text{ where} \\ & \quad \quad \forall \sigma_2. \theta(\sigma_2) = \mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma, \theta_1, \sigma_3 \text{ where} \\ & \quad \quad \quad \text{let } (r_1, \tau_1, \sigma_3) = \sigma_2.instantiateClass(r) \text{ in} \\ & \quad \quad \quad \forall \sigma_4. \theta_1(\sigma_4) = \kappa(r_1, \tau_1, \sigma_4) \\ | \mathcal{E}[\{ \langle \text{BlockStmtList} \rangle^? \}] \gamma \kappa \sigma = & \\ & \mathcal{E}[\mathbf{super} ()] \gamma \kappa_1 \sigma \text{ where} \\ & \quad \forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) = \theta(\sigma_1) \text{ where} \\ & \quad \quad \forall \sigma_2. \theta(\sigma_2) = \mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma, \theta_1, \sigma_3 \text{ where} \\ & \quad \quad \quad \text{let } (r_1, \tau_1, \sigma_3) = \sigma_2.instantiateClass(r) \text{ in} \\ & \quad \quad \quad \forall \sigma_4. \theta_1(\sigma_4) = \kappa(r_1, \tau_1, \sigma_4) \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{ExplConstrInv} \rangle] \gamma \kappa \sigma ::= & \\ & \mathcal{E}[\mathbf{this} (\langle \text{ArgList} \rangle^?) ;] \gamma \kappa \sigma \\ | \mathcal{E}[\mathbf{super} (\langle \text{ArgList} \rangle^?) ;] \gamma \kappa \sigma & \end{aligned}$$

4.12 Blocks and Statements

In this section, we present the semantics for blocks and statements in the Java language. We differ from the grammar in the JLS [1], by not presenting any of the statements associated with the *No Short If* constructs, used in the JLS to avoid syntactic ambiguity with dangling else clauses. The semantics for all of these clauses can be easily derived from the semantics presented here.

Blocks. A block consists of an optional sequence of block statements within a pair of braces. Note here, that if there are no block statements, the semantics of the block are equivalent to $\theta(\gamma, \sigma)$.

$$\begin{aligned} \mathcal{C}[\langle \text{Block} \rangle] \gamma \theta \sigma ::= & \\ & \mathcal{C}[\{ \langle \text{BlockStmtList} \rangle^? \}] \gamma \theta \sigma = \mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma \theta \sigma \end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{C}[\langle \text{BlockStmt} \rangle] \gamma \theta \sigma \\
&| \mathcal{C}[\langle \text{BlockStmtList}_1 \rangle \langle \text{BlockStmt} \rangle] \gamma \theta \sigma = \mathcal{C}[\langle \text{BlockStmtList}_1 \rangle] \gamma \theta_1 \sigma \text{ where} \\
&\quad \forall \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) = \mathcal{C}[\langle \text{BlockStmt} \rangle] \gamma_1 \theta \sigma_1
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{BlockStmt} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{C}[\langle \text{LocalVarDeclStmt} \rangle] \gamma \theta \sigma \\
&| \mathcal{C}[\langle \text{Stmt} \rangle] \gamma \theta \sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{LocalVarDeclStmt} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{C}[\langle \text{LocalVarDecl} \rangle ;] \gamma \theta \sigma
\end{aligned}$$

Local Variable Declarations. These modify both the environment and the store (local store), by creating a new semantic entity. As such, a local variable declaration will continue program execution with these new attributes.

$$\begin{aligned}
\mathcal{C}[\langle \text{LocalVarDecl} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{D}[\langle \text{Type} \rangle \langle \text{VarDeclList} \rangle] \gamma \theta \sigma = \mathcal{D}[\langle \text{VarDeclList} \rangle] \gamma \delta \sigma \text{ where} \\
&\quad \forall d, \gamma_1, \sigma_1. \delta(d, \gamma_1, \sigma_1) = \theta(\gamma_2, \sigma_1) \text{ where} \\
&\quad \gamma_2 = \gamma_1[\&Type \leftarrow \mathcal{T}[\langle \text{Type} \rangle] \gamma, \&varType \leftarrow \text{“Local”}]
\end{aligned}$$

Empty, Labeled and Expression Statements. These statements are basic primitive statements of the Java language. The empty statement consists solely of a single semicolon and semantically continues operation as if nothing happened. The labeled statement modifies the environment to contain an identifier, Id that refers to the current statement. The environment maintains the semantic evaluation of the statement as a function of the current statement, parameterized by possibly new environment and store. Note that the environment of the statement contains a reference to the label, Id , but upon completion of execution, that label is removed from the environment. The expression statement evaluates the expression using the semantic function for expressions, discarding any returned value or type, and continues execution using the possibly modified store. Note that we have simplified an expression statement to consist of any expression, although this is not strictly true. In the JLS [1], the grammar restricts expression statements to a list of possible expressions. We take liberty with our assumption of syntactically correct programs to simplify the grammar here.

$$\begin{aligned}
\mathcal{C}[\langle \text{EmptyStmt} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{C}[\langle ; \rangle] \gamma \theta \sigma = \theta(\gamma, \sigma)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{LabeledStmt} \rangle] \gamma \theta \sigma &::= \\
&\mathcal{C}[\langle Id : \langle \text{Stmt} \rangle \rangle] \gamma \theta \sigma = \mathcal{C}[\langle \text{Stmt} \rangle] \gamma_1 \theta_1 \sigma \text{ where} \\
&\quad \gamma_1 = \gamma[Id \leftarrow \theta_2] \text{ where} \\
&\quad \forall \gamma_2, \sigma_2. \theta_2(\gamma_2, \sigma_2) = \mathcal{C}[\langle \text{Stmt} \rangle] \gamma_2 \theta_1 \sigma_2 \\
&\quad \forall \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) = \theta(\gamma, \sigma_1)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{ExprStmt} \rangle ;] \gamma \theta \sigma &::= \\
&\mathcal{C}[\langle \text{Expr} \rangle] \gamma \theta \sigma = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\
&\quad \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) = \theta(\gamma, \sigma_1)
\end{aligned}$$

If Statements. The if statement has two forms, one with and one without an else clause. The if statement executes the expression first, possibly modifying the store, and then behaves as the first statement if the expression is true. If the expression is false it either continues execution or behaves as the statement following the else clause.

$$\begin{aligned} \mathcal{C}[\langle \text{IfStmt} \rangle] \gamma \theta \sigma &::= \\ \mathcal{C}[\text{if} (\langle \text{Expr} \rangle) \langle \text{Stmt} \rangle] \gamma \theta \sigma &= \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\ \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) &= \\ \text{if } (r == \mathbf{true}) & \\ \mathcal{C}[\langle \text{Stmt} \rangle] \gamma \theta \sigma_1 & \\ \text{else} & \\ \theta(\gamma, \sigma_1) & \\ \text{endif} & \end{aligned}$$

$$\begin{aligned} \mathcal{C}[\langle \text{IfElseStmt} \rangle] \gamma \theta \sigma &::= \\ \mathcal{C}[\text{if} (\langle \text{Expr} \rangle) \langle \text{Stmt}_1 \rangle \mathbf{else} \text{ Stmt}_2] \gamma \theta \sigma &= \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\ \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) &= \\ \text{if } (r == \mathbf{true}) & \\ \mathcal{C}[\langle \text{Stmt}_1 \rangle] \gamma \theta \sigma_1 & \\ \text{else} & \\ \mathcal{C}[\langle \text{Stmt}_2 \rangle] \gamma \theta \sigma_1 & \\ \text{endif} & \end{aligned}$$

The Switch Statement. The Java switch statement presents a few problems for the design of a denotational semantics. The problems we found and their resolution are discussed below. The approach we took involves modification of the environment to provide additional information to subsequent semantic functions. This modification is in the form of auxiliary variables. Note that these variables must be restored to their previous values upon completion of the switch statement to permit the correct evaluation of nested switch statements.

- The data value obtained upon execution of the switch statement determines which case to execute. Thus this value must be carried along through the semantic functions until it is utilized. We decided to maintain the value in the environment under the auxiliary variable name *&switchExpr*.
- Once a case label has been found to match the switch expression, all subsequent switch block statements are to be executed. Thus, the semantic meaning of these statements is dependent on whether or not a case label matched the switch expression. We decided to maintain a boolean flag, *&caseFound*, in the environment to indicate whether or not a match has been found.
- The default switch case label may occur any place a case label may occur. If no case label matches the switch expression, the meaning of the switch statement is the meaning of all switch block statements that follow the default label. The problem is that not only do we have to inform the semantic evaluation functions that a default label has been found, but the functions also have to allow for the existence of a matching case label occurring after the default label. The first problem is resolved with the boolean flag

&defaultFound, which operates the same as the *&caseFound* flag. The other problem is resolved using the *&caseCont* variable which records the environment, store and continuation parameters for the switch statement.

- Unlabelled break statement may occur within a switch statement. The intent of this statement is to terminate execution of the switch statement. As such, the break information is stored in the environment.

$$\begin{aligned}
\mathcal{C}[\langle \text{SwitchStmt} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\text{switch} (\langle \text{Expr} \rangle) \langle \text{SwitchBlock} \rangle] \gamma \theta \sigma &= \\
\mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} & \\
\forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) = \mathcal{C}[\langle \text{SwitchBlock} \rangle] \gamma_1 \theta_1 \sigma_1 \text{ where} & \\
\gamma_1 = \gamma[\&switchExpr \leftarrow r, \&caseFound \leftarrow \text{false}, & \\
\&defaultFound \leftarrow \text{false}, \&caseCont \leftarrow (\gamma, \theta_1, \sigma_1), & \\
\&break \leftarrow \theta_1] \text{ and} & \\
\forall \gamma_1, \sigma_2. \theta_1(\gamma_1, \sigma_2) = \theta(\gamma, \sigma_2) &
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{SwitchBlock} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\{ \langle \text{SwitchBlockStmtList} \rangle^? \langle \text{SwitchLabelList} \rangle^? \}] \gamma \theta \sigma &= \\
\mathcal{C}[\langle \text{SwitchBlockStmtList} \rangle^?] \gamma \theta_1 \sigma \text{ where} & \\
\forall \gamma_1, \sigma_2. \theta_1(\gamma_1, \sigma_2) = \mathcal{C}[\langle \text{SwitchLabelList} \rangle] \gamma_1 \theta \sigma_2 &
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{SwitchBlockStmtList} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{SwitchBlockStmt} \rangle] \gamma \theta \sigma & \\
| \mathcal{C}[\langle \text{SwitchBlockStmtList}_1 \rangle \langle \text{SwitchBlockStmt} \rangle] \gamma \theta \sigma = & \\
\mathcal{C}[\langle \text{SwitchBlockStmtList} \rangle^?] \gamma \theta_1 \sigma \text{ where} & \\
\forall \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) = \mathcal{C}[\langle \text{SwitchBlockStmt} \rangle^?] \gamma_1 \theta \sigma_1 &
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{SwitchBlockStmt} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{SwitchLabelList} \rangle \langle \text{BlockStmtList} \rangle] \gamma \theta \sigma = & \\
\text{if } (\gamma.\text{getValue}(\&caseFound) == \text{true}) & \\
\mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma \theta \sigma & \\
\text{else} & \\
\mathcal{C}[\langle \text{SwitchLabelList} \rangle] \gamma \theta_1 \sigma \text{ where} & \\
\forall \gamma_1, \sigma_1. \theta_1(\gamma_1 \sigma_1) = & \\
\text{if } (\gamma_1(\&caseFound) == \text{true}) & \\
\mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma_1(\&caseCont) & \\
\text{else if } \gamma_1(\&defaultFound) & \\
\mathcal{C}[\langle \text{BlockStmtList} \rangle] \gamma_1 \theta \sigma_1 & \\
\text{else} & \\
\theta(\gamma_1, \sigma_1) & \\
\text{endif} & \\
\text{endif} &
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{SwitchLabelList} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{SwitchLabel} \rangle] \gamma \theta \sigma & \\
| \mathcal{C}[\langle \text{SwitchLabelList}_1 \rangle \langle \text{SwitchLabel} \rangle] \gamma \theta \sigma = & \\
\mathcal{C}[\langle \text{SwitchLabelList}_1 \rangle] \gamma \theta_1 \sigma \text{ where} & \\
\forall \gamma_1, \sigma_1. \theta_1(\gamma_1 \sigma_1) = \mathcal{C}[\langle \text{SwitchLabel} \rangle] \gamma_1 \theta \sigma_1 &
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{SwitchLabel} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{case } \langle \text{Const Expr} \rangle \rangle] \gamma \theta \sigma &= \mathcal{E}[\langle \text{Const Expr} \rangle] \gamma \kappa \sigma \text{ where} \\
\forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) &= \\
&\text{if } (\text{fst}(r) == \gamma[\&\text{switchExpr}]) \\
&\quad \theta(\gamma[\&\text{caseFound} \leftarrow \text{true}], \sigma_1) \\
&\text{else} \\
&\quad \theta(\gamma, \sigma_1) \\
&\text{endif} \\
| \mathcal{C}[\langle \text{default } \rangle] \gamma \theta \sigma &= \theta(\gamma[\&\text{defaultFound} \leftarrow \text{true}], \sigma)
\end{aligned}$$

Looping Statements. The Java language looping constructs, the while, do and for statements, are similar to the looping constructs of other languages. And as such, they cause difficulty for the writing of denotational semantics. Two different approaches to specifying the semantics of loops have been presented in the literature, the fixpoint approach [4] and the recursive definition approach [5]. We have defined the do-statement and the for-statement in terms of the while statement. Note that the expression in the for statement is optional. In the case where it is not present, the semantics need to assume that the result is always true; we have divided this case into two separate syntactic forms for clarity.

$$\begin{aligned}
\mathcal{C}[\langle \text{WhileStmt} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{while } (\langle \text{Expr} \rangle) \langle \text{Statement} \rangle \rangle] \gamma \theta \sigma &= \theta_1(\gamma[\&\text{break} \leftarrow \theta], \sigma) \text{ where} \\
\text{rec, } \forall \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) &= \mathcal{E}[\langle \text{Expr} \rangle] \gamma_1 \kappa \sigma \text{ where} \\
\forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) &= \\
&\text{if } (r == \text{true}) \\
&\quad \mathcal{C}[\langle \text{Statement} \rangle] \gamma_1 \theta_1 \sigma_1 \\
&\text{else} \\
&\quad \theta(\gamma, \sigma_1) \\
&\text{endif}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{DoStmt} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{do } \langle \text{Statement} \rangle \text{ while } (\langle \text{Expr} \rangle) \rangle] \gamma \theta \sigma &= \\
\mathcal{C}[\langle \text{Statement} \rangle ; \text{while } (\langle \text{Expr} \rangle) \langle \text{Statement} \rangle] \gamma \theta \sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{ForStmt} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{for } (\langle \text{ForInit} \rangle^? ; \langle \text{ForUpdate} \rangle^?) \langle \text{Statement} \rangle \rangle] \gamma \theta \sigma &= \\
\mathcal{C}[\langle \text{ForInit} \rangle ; \text{while } (\text{true}) \langle \text{Statement} \rangle ; \langle \text{ForUpdate} \rangle] \gamma \theta \sigma & \\
| \mathcal{C}[\langle \text{for } (\langle \text{ForInit} \rangle^? ; \langle \text{Expr} \rangle ; \langle \text{ForUpdate} \rangle^?) \langle \text{Statement} \rangle \rangle] \gamma \theta \sigma &= \\
\mathcal{C}[\langle \text{ForInit} \rangle ; \text{while } (\langle \text{Expr} \rangle) \langle \text{Statement} \rangle ; \langle \text{ForUpdate} \rangle] \gamma \theta \sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{ForInit} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{StmtExprList} \rangle] \gamma \theta \sigma & \\
| \mathcal{C}[\langle \text{LocalVarDecl} \rangle] \gamma \theta \sigma &
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{ForUpdate} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{StmtExprList} \rangle] \gamma \theta \sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\langle \text{StmtExprList} \rangle] \gamma \theta \sigma &::= \\
\mathcal{C}[\langle \text{ExprStmt} \rangle] \gamma \theta \sigma
\end{aligned}$$

$$\begin{aligned} & | \mathcal{C}[\langle \text{StmtExprList}_1 \rangle, \langle \text{ExprStmt} \rangle] \gamma \theta \sigma = \mathcal{C}[\langle \text{StmtExprList}_1 \rangle] \gamma \theta_1 \sigma \text{ where} \\ & \quad \forall \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) = \mathcal{C}[\langle \text{ExprStmt} \rangle] \gamma_1 \theta \sigma_1 \end{aligned}$$

Misc. The following semantic functions define the behavior of the miscellaneous syntactic commands in the Java language. The expression statement list involves evaluation of list of expression, with the result values discarded. The break, continue, return, and throw statements all evaluate their parameters and then look up the corresponding continuation in the environment. The meaning of the rest of the program is based on this continuation. The synchronized command is ignored in these semantics since we do not specify concurrency.

$$\begin{aligned} & \mathcal{C}[\langle \text{ExprStmtList} \rangle] \gamma \theta \sigma ::= \\ & \quad \mathcal{C}[\langle \text{ExprStmtList}_1 \rangle, \langle \text{ExprStmt} \rangle] \gamma \theta \sigma = \mathcal{C}[\langle \text{ExprStmtList}_1 \rangle] \gamma \theta_1 \sigma \text{ where} \\ & \quad \forall \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) = \mathcal{E}[\langle \text{ExprStmt} \rangle] \gamma_1 \kappa \sigma_1 \text{ where} \\ & \quad \quad \forall r, \tau, \sigma_2. \kappa(r, \tau, \sigma_2) = \theta_1(\gamma_1, \sigma_2) \end{aligned}$$

$$\begin{aligned} & \mathcal{C}[\langle \text{BreakStmt} \rangle] \gamma \theta \sigma ::= \\ & \quad \mathcal{C}[\mathbf{break};] \gamma \theta \sigma = \theta_1(\gamma \sigma) \text{ where} \\ & \quad \quad \theta_1 = \gamma.\text{getComCont}(\&\text{break}) \\ & | \mathcal{C}[\mathbf{break} \langle \text{Id} \rangle;] \gamma \theta \sigma = \theta_1(\gamma \sigma) \text{ where} \\ & \quad \quad \theta_1 = \gamma.\text{getComCont}(\&\text{break}) (\text{fst}(\mathcal{V}[\text{Id}]\gamma)) \end{aligned}$$

$$\begin{aligned} & \mathcal{C}[\langle \text{ContStmt} \rangle] \gamma \theta \sigma ::= \\ & \quad \mathcal{C}[\mathbf{continue};] \gamma \theta \sigma = \theta_1(\gamma \sigma) \text{ where} \\ & \quad \quad \theta_1 = \gamma.\text{getComCont}(\&\text{continue}) \\ & | \mathcal{C}[\mathbf{continue} \langle \text{Id} \rangle;] \gamma \theta \sigma = \theta_1(\gamma \sigma) \text{ where} \\ & \quad \quad \theta_1 = \gamma.\text{getComCont}(\&\text{continue}) \text{fst}(\mathcal{V}[\text{Id}]\gamma) \end{aligned}$$

$$\begin{aligned} & \mathcal{C}[\langle \text{RetStmt} \rangle] \gamma \theta \sigma ::= \\ & \quad \mathcal{C}[\mathbf{return};] \gamma \theta \sigma = \gamma.\text{getComCont}(\&\text{return}) \\ & | \mathcal{C}[\mathbf{return} \langle \text{Expr} \rangle;] \gamma \theta \sigma = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\ & \quad \forall r, \tau, \sigma. \kappa(r, \tau, \sigma) = \theta_1(\sigma_1) \text{ where} \\ & \quad \quad \theta_1 = \gamma.\text{getComCont}(\&\text{return}) \text{ and} \\ & \quad \quad \tau_1 = \gamma[\&\text{returnType}] \text{ and} \\ & \quad \quad r_1 = \mathbf{promote}(\tau_1, (r, \tau)) \text{ and} \\ & \quad \quad \sigma_1 = \sigma[\&\text{returnVal} \leftarrow r_1] \end{aligned}$$

$$\begin{aligned} & \mathcal{C}[\langle \text{ThrowStmt} \rangle] \gamma \theta \sigma ::= \\ & \quad \mathcal{C}[\mathbf{throw} \langle \text{Expr} \rangle;] = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\ & \quad \text{forall } r, \tau, \gamma_1, \sigma_2. \kappa(r, \tau, \gamma_1, \text{sto}_1) = \theta_1(\gamma_2, \sigma_1) \text{ where} \\ & \quad \quad \gamma_2 = \gamma_1[\&\text{thrown} \leftarrow (r, \tau)] \text{ and} \\ & \quad \quad \theta_1 = \gamma.[\&\text{throw}] \end{aligned}$$

$$\begin{aligned} & \mathcal{C}[\langle \text{SynchStmt} \rangle] \gamma \theta \sigma ::= \\ & \quad \mathcal{C}[\mathbf{synchronized} (\langle \text{Expr} \rangle) \langle \text{Block} \rangle] \gamma \theta \sigma = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\ & \quad \quad \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) = \mathcal{C}[\langle \text{Block} \rangle] \gamma \theta \sigma_1 \end{aligned}$$

The try-catch statements of the Java language are an important aspect of the language for error control. The following semantic functions capture the

meaning of these statements. A try block is executed until a throw command is executed, at that point the execution continues based on the continuation stored in the environment. This continuation consists of the execution of the finally block followed by evaluation of the catch parameter. If the thrown exception matches the formal parameter, the catch clause is executed and the program continues using the continuation from the commands following the try block. If none of the catch clauses match, then the throw propagates on up.

$$\begin{aligned} \mathcal{C}[\langle \text{TryStmt} \rangle] \gamma \theta \sigma &::= \\ \mathcal{C}[\langle \text{try } \langle \text{Block} \rangle \langle \text{Catches} \rangle] \gamma \theta \sigma &= \mathcal{C}[\langle \text{Block} \rangle] \gamma_1 \theta_1 \sigma \text{ where} \\ \gamma_1 &= \gamma[\&throw \leftarrow \theta_2] \text{ and} \\ \forall \gamma_2, \sigma_2. \theta_1(\gamma_2, \sigma_2) &= \theta(\gamma, \sigma_2) \text{ and} \\ \forall \gamma_2, \sigma_2. \theta_2(\gamma_2, \sigma_2) &= \mathcal{C}[\langle \text{Catches} \rangle] \gamma \theta_3 \sigma_2 \text{ where} \\ \forall \gamma_3, \sigma_3. \theta_3(\gamma_3, \sigma_3) &= \\ \text{if } (\gamma_3(\&thrown) == (\text{null}, \text{"V"})) &\text{ then} \\ \theta(\gamma_3, \sigma_3) & \\ \text{else} & \\ (\gamma_3.[\&throw])(\gamma_3, \sigma_3) & \\ \text{endif} & \\ | \mathcal{C}[\langle \text{try } \langle \text{Block} \rangle \langle \text{Catches} \rangle? \langle \text{Finally} \rangle] \gamma \theta \sigma &= \mathcal{C}[\langle \text{Block} \rangle] \gamma_1 \theta_1 \sigma \text{ where} \\ \gamma_1 &= \gamma[\&throw \leftarrow \theta_2] \text{ and} \\ \forall \gamma_2, \sigma_2. \theta_1(\gamma_2, \sigma_2) &= \mathcal{C}[\langle \text{finally} \rangle] \gamma \theta \sigma_2 \text{ and} \\ \forall \gamma_2, \sigma_2. \theta_2(\gamma_2, \sigma_2) &= \mathcal{C}[\langle \text{finally} \rangle] \gamma \theta_3 \sigma_2 \text{ where} \\ \forall \gamma_3, \sigma_3. \theta_3(\gamma_3, \sigma_3) &= \mathcal{C}[\langle \text{Catches} \rangle] \gamma_3 \theta_4 \sigma_3 \text{ where} \\ \forall \gamma_4, \sigma_4. \theta_4(\gamma_4, \sigma_4) &= \\ \text{if } (\gamma_4(\&thrown) == (\text{null}, \text{"V"})) &\text{ then} \\ \theta(\gamma_4, \sigma_4) & \\ \text{else} & \\ (\gamma_4.[\&throw])(\gamma_4, \sigma_4) & \\ \text{endif} & \\ \mathcal{C}[\langle \text{Catches} \rangle] \gamma \theta \sigma &::= \\ \mathcal{C}[\langle \text{CatchClause} \rangle] \gamma \theta \sigma & \\ | \mathcal{C}[\langle \text{Catches}_1 \rangle \langle \text{CatchClause} \rangle] \gamma \theta \sigma &= \mathcal{C}[\langle \text{Catches}_1 \rangle] \gamma \theta_1 \sigma \text{ where} \\ \text{forall } \gamma_1, \sigma_1. \theta_1(\gamma_1, \sigma_1) &= \mathcal{C}[\langle \text{CatchClause} \rangle] \gamma_1 \theta \sigma_1 \\ \mathcal{C}[\langle \text{CatchClause} \rangle] \gamma \theta \sigma &::= \\ \mathcal{C}[\langle \text{catch } (\langle \text{FormalParam} \rangle) \langle \text{Block} \rangle] \gamma \theta \sigma &= \\ \text{let } (r, \tau) = \gamma(\&thrown) \text{ and} & \\ (e, \tau_1 = \mathcal{V}[\langle \text{FormalParam} \rangle]) \text{ in} & \\ \text{if } (\tau == \tau_1) \text{ then} & \\ \mathcal{C}[\langle \text{Block} \rangle] \gamma_1 \theta \sigma_1 \text{ where} & \\ \gamma_1 = \gamma[\&thrown \leftarrow (\text{null}, \text{"V"})] \text{ and} & \\ \sigma_1 = \sigma[\gamma(e) \leftarrow r] & \\ \text{else} & \\ \theta(\gamma, \sigma) & \\ \text{endif} & \\ \mathcal{C}[\langle \text{Finally} \rangle] \gamma \theta \sigma &::= \end{aligned}$$

$$\mathcal{C}[\text{finally } \langle \text{Block} \rangle] \gamma \theta \sigma = \mathcal{C}[\langle \text{Block} \rangle] \gamma \theta \sigma$$

4.13 Expressions

Expressions in Java return either values or variables. In these semantics we have broken these into two categories, handled by different semantic functions, $\mathcal{E}[\![]]$ for values and $\mathcal{L}[\![]]$ for variables. The first two syntactic expressions denote constant expressions (which must return a value) and general expressions (which also return values). Note that restriction that constant expressions return constant values is a compile-time check and thus is not represented in these semantics.

$$\begin{aligned} \mathcal{E}[\langle \text{Constant Expr} \rangle] \gamma \kappa \sigma &::= \\ \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma &::= \\ \mathcal{E}[\langle \text{Assign Expr} \rangle] \gamma \kappa \sigma \end{aligned}$$

Assignment Expressions: There are several assignment operators in Java besides the simple assignment. According to the JLS [1] the compound assignment $E_1 \text{ op} = E_2$ is equivalent to $E_1 = (T)((E_1) \text{ op}(E_2))$ where T is the type of E_1 and the expression E_1 is evaluated only once. In the semantic model, we evaluate E_1 once to obtain its memory location for assignment in the store, and use that location to determine the value of the expression for the operation. This evaluation requires the expression to return a variable (actually a variable location for use by the store) as opposed to a value. To indicate this return type we use the location $\mathcal{L}[\![]]$ semantic functions.

$$\begin{aligned} \mathcal{E}[\langle \text{Assign Expr} \rangle] \gamma \kappa \sigma &::= \\ \mathcal{E}[\langle \text{Cond Expr} \rangle] \gamma \kappa \sigma \\ | \mathcal{E}[\langle \text{Assign} \rangle] \gamma \kappa \sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{Assign} \rangle] \gamma \kappa \sigma &::= \\ \mathcal{E}[\langle \text{LHS} \rangle \langle \text{Assign Op} \rangle \langle \text{Assign Expr} \rangle] \gamma \kappa \sigma = \\ \text{if } (\text{Assign Op} == \text{'+'}) & \\ \mathcal{L}[\langle \text{LHS} \rangle] \gamma \alpha \sigma \text{ where} & \\ \forall r_1, \tau_1, l, \sigma_1. \alpha(r_1, \tau_1, l, \sigma_1) = \mathcal{E}[\langle \text{Assign Expr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} & \\ \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = & \\ \text{let } \tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2) \text{ and} & \\ d = \mathbf{cast}(\tau_1, (\mathbf{promote}(\tau, (r_1, \tau_1)) +_{\tau} \mathbf{promote}(\tau, (r_2, \tau_2)), \tau)) \text{ in} & \\ \kappa(d, \tau_1, \sigma_2[l \leftarrow d]) & \end{aligned}$$

similar for $- =, * =, \% =, \& =, \wedge =, | =$

where the meaning of op_{τ} is defined in the section on numeric expressions

$$\begin{aligned} \text{else if } (\text{Assign Op} == \text{'/'}) & \\ \mathcal{L}[\langle \text{LHS} \rangle] \gamma \alpha \sigma \text{ where} & \\ \forall r_1, \tau_1, l, \sigma_1. \alpha(r_1, \tau_1, l, \sigma_1) = \mathcal{E}[\langle \text{Assign Expr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} & \\ \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = & \end{aligned}$$

```

let  $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$  in
if ( $r_2 = 0 \wedge (\tau = \text{"I"} \vee \tau = \text{"L"})$ )
   $\theta(\gamma_1, \sigma_3)$  where
   $\theta = \gamma.[\&throw]$  and
   $\sigma_3, r_3, \tau_3 = \sigma.mkException(ArithmeticException)$  and
   $\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$ 
else
  let  $d = \mathbf{cast}(\tau_1, (\mathbf{promote}(\tau, (r_1, \tau_1)) /_{\tau} \mathbf{promote}(\tau, (r_2, \tau_2))), \tau)$  in
  ( $\kappa(d, \tau_1, \sigma_2[l \leftarrow d])$ )
endif
else if (AssignOp == '<<=')
 $\mathcal{L}[\langle \text{LHS} \rangle] \gamma \alpha \sigma$  where
 $\forall r_1, \tau_1, l, \sigma_1. \alpha(r_2, \tau_1, l, \sigma_1) = \mathcal{E}[\langle \text{AssignExpr} \rangle] \gamma \kappa_2 \sigma_1$  where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) =$ 
  let  $\tau'_1 = \mathbf{unaryPromoteType}(\tau_1)$  and
   $\tau'_2 = \mathbf{unaryPromoteType}(\tau_2)$  and
   $r'_1 = \mathbf{promote}(\tau'_1, (r_1, \tau_1))$  and
   $r'_2 = \mathbf{promote}(\tau'_2, (r_2, \tau_2))$  and
   $d = (\mathbf{leftShift}((r'_1, \tau'_1), (r'_2, \tau'_2)))$  in
   $\kappa(d, \tau_1, \sigma_2[l \leftarrow d])$ 

similar for  $>>=, >>>=$ 
where the meaning of  $op_{\tau}$  is defined in the section on numeric expressions

else if (AssignOp == '=')
 $\mathcal{L}[\langle \text{LHS} \rangle] \gamma \alpha \sigma$  where
 $\forall r_1, \tau_1, l, \sigma_1. \alpha(r_1, \tau_1, l, \sigma_1) = \mathcal{E}[\langle \text{AssignExpr} \rangle] \gamma \kappa_2 \sigma_1$  where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) =$ 
  if ( $r_1 == \text{null}$ )
     $\theta(\gamma_1, \sigma_3)$  where
     $\theta = \gamma.[\&throw]$  and
     $\sigma_3, r_3, \tau_3 = \sigma.mkException(NullPointerException)$  and
     $\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$ 
  else if ( $r_1 == \text{OutOfBounds}$ )
     $\theta(\gamma_1, \sigma_3)$  where
     $\theta = \gamma.[\&throw]$  and
     $\sigma_3, r_3, \tau_3 = \sigma.mkException(IndexOutOfBoundsException)$  and
     $\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$ 
  else if not ( $\tau_2 <_{\mathcal{T}} \tau_1$ ) then
     $\theta(\gamma_1, \sigma_3)$  where
     $\theta = \gamma.[\&throw]$  and
     $\sigma_3, r_3, \tau_3 = \sigma.mkException(ArrayStoreException)$  and
     $\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$ 
  else
     $\kappa(r_2, \tau_1, \sigma_2[l \leftarrow \mathbf{promote}(\tau_1, (r_2, \tau_2))])$ 
endif

endif

<AssignOp> ::=
=
```

```

| * =
| / =
| % =
| + =
| - =
| <<=
| >>=
| >>>=
| & =
| ^ =
| | =

```

$$\begin{aligned} \mathcal{L}[\langle \text{LHS} \rangle] \gamma \alpha \sigma ::= & \\ & \mathcal{L}[\langle \text{Name} \rangle] \gamma \alpha \sigma \\ & | \mathcal{L}[\langle \text{FieldAccess} \rangle] \gamma \alpha \sigma \\ & | \mathcal{L}[\langle \text{ArrayAccess} \rangle] \gamma \alpha \sigma \end{aligned}$$

Conditional Expressions. The conditional expressions (operator: ?) of the Java language are the only expressions that do not guarantee that all subexpressions are evaluated. The regular conditional expression, is a choice operation that executes the second or third subexpression based on the boolean result of the first subexpression. The return type of the expression is based on the type of the two possible resultant subexpressions. The conditional-or expression (operator ||) and the conditional-and expression (operator &&) are *short-circuit* boolean expressions that only evaluate the second subexpression if the result of the first subexpression does not determine the result of the expression (i.e., short circuits on **true** for or and **false** for and).

$$\begin{aligned} \mathcal{E}[\langle \text{CondExpr} \rangle] \gamma \kappa \sigma ::= & \\ & \mathcal{E}[\langle \text{CondOrExpr} \rangle] \gamma \kappa \sigma \\ & | \mathcal{E}[\langle \text{CondOrExpr} \rangle ? \langle \text{Expr} \rangle : \langle \text{CondExpr}_1 \rangle] \gamma \kappa \sigma = \\ & \quad \mathcal{E}[\langle \text{CondOrExpr} \rangle] \gamma \kappa_1 \sigma \text{ where} \\ & \quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r, \tau, \sigma_1) = \\ & \quad \text{if } (r_1 == \mathbf{true}) \\ & \quad \quad \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa_2 \sigma_1 \\ & \quad \text{else} \\ & \quad \quad \mathcal{E}[\langle \text{CondOrExpr}_1 \rangle] \gamma \kappa_2 \sigma_1 \\ & \quad \text{endif} \\ & \quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \\ & \quad \kappa(\mathbf{promote}(\tau, (r_2, \tau_2)), \tau, \sigma_2) \text{ where} \\ & \quad \tau = env.condTypeOf(\tau_t, v_t, \tau_f, v_f) \text{ and} \\ & \quad v_t = \text{compile-time value of } \langle \text{Expr} \rangle \text{ and} \\ & \quad \tau_t = \text{type of } \langle \text{Expr} \rangle \text{ and} \\ & \quad \tau_f = \text{type of } \langle \text{CondOrExpr}_1 \rangle \text{ and} \\ & \quad v_f = \text{compile-time value of } \langle \text{CondOrExpr}_1 \rangle \end{aligned}$$

we compute the result of $r_1 <_{\tau} r_2$ using the following:

$$\gamma.condTypeOf(\tau_t, v_t, \tau_f, v_f) =$$

```

if ( $\tau_t == \tau_f$ )
   $\tau_t$ 
else if (isNumeric( $\tau_t$ ) and isNumeric( $\tau_f$ ))
  if (( $\tau_t == \text{"B"}$  and  $\tau_f == \text{"S"}$ ) or
      (( $\tau_f == \text{"B"}$  and  $\tau_t == \text{"S"}$ ))
      "S")
  else if ( $\tau_t \in [\text{"B"}, \text{"S"}, \text{"C"}]$  and  $v_f \in \tau_t$ )
   $\tau_t$ 
  else if ( $\tau_f \in [\text{"B"}, \text{"S"}, \text{"C"}]$  and  $v_t \in \tau_f$ )
   $\tau_f$ 
  else
    binaryPromotionType( $\tau_t, \tau_f$ )
  end if
else if ( $\gamma.assnCompatible(\tau_t, \tau_f)$ )
   $\tau_f$ 
else
   $\tau_t$ 
endif

 $\mathcal{E}[\langle \text{CondOrExpr} \rangle] \gamma \kappa \sigma ::=$ 
   $\mathcal{E}[\langle \text{CondAndExpr} \rangle] \gamma \kappa \sigma$ 
  |  $\mathcal{E}[\langle \text{CondOrExpr}_1 \rangle \ || \ \langle \text{CondAndExpr} \rangle] \gamma \kappa \sigma =$ 
   $\mathcal{E}[\langle \text{CondOrExpr}_1 \rangle] \gamma \kappa_1 \sigma$  where
     $\forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) =$ 
    if ( $r == \text{true}$ )
       $\kappa(r, \tau, \sigma_1)$ 
    else
       $\mathcal{E}[\langle \text{CondAndExpr} \rangle] \gamma \kappa \sigma_1$ 
    endif

 $\mathcal{E}[\langle \text{CondAndExpr} \rangle] \gamma \kappa \sigma ::=$ 
   $\mathcal{E}[\langle \text{IncOrExpr} \rangle] \gamma \kappa \sigma$ 
  |  $\mathcal{E}[\langle \text{CondAndExpr}_1 \rangle \ \&\& \ \langle \text{IncOrExpr} \rangle] \gamma \kappa \sigma =$ 
   $\mathcal{E}[\langle \text{CondAndExpr}_1 \rangle] \gamma \kappa_1 \sigma$  where
     $\forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) =$ 
    if ( $r == \text{true}$ )
       $\kappa(r, \tau, \sigma_1)$ 
    else
       $\mathcal{E}[\langle \text{IncOrExpr} \rangle] \gamma \kappa \sigma_1$ 
    endif

```

Bitwise and Boolean Expressions . The following expressions all return boolean results, with the exception of the first three (and, or and xor) which perform bitwise operations on integral operands and logical operations on boolean operands. The comparison expressions can work with any operands of compatible types and thus require a more extensive definition. The results of these operations are rather complex for floating point values, and have been defined in tables to simplify the presentation. Shift and comparison operations that a similar have been removed from this presentation for space consideration.

$$\begin{aligned}
\mathcal{E}[\langle \text{IncOrExpr} \rangle] \gamma \kappa \sigma ::= & \\
& \mathcal{E}[\langle \text{XORExpr} \rangle] \gamma \kappa \sigma \\
& | \mathcal{E}[\langle \text{IncOrExpr}_1 \rangle | \langle \text{XORExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{IncOrExpr}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\
& \quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{XORExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\
& \quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(r_1 \text{ or}_\perp r_2, \tau, \sigma_2) \text{ where} \\
& \quad \text{if } (\tau_1 == \text{"Z"}) \\
& \quad \quad \tau = \text{"Z"} \\
& \quad \text{else} \\
& \quad \quad \tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2) \\
& \text{endif}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\langle \text{XORExpr} \rangle] \gamma \kappa \sigma ::= & \\
& \mathcal{E}[\langle \text{AndExpr} \rangle] \gamma \kappa \sigma \\
& | \mathcal{E}[\langle \text{XORExpr}_1 \rangle \wedge \langle \text{AndExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{XORExpr}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\
& \quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{AndExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\
& \quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(r_1 \text{ xor}_\perp r_2, \tau, \sigma_2) \text{ where} \\
& \quad \text{if } (\tau_1 == \text{"Z"}) \\
& \quad \quad \tau = \text{"Z"} \\
& \quad \text{else} \\
& \quad \quad \tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2) \\
& \text{endif}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\langle \text{AndExpr} \rangle] \gamma \kappa \sigma ::= & \\
& \mathcal{E}[\langle \text{EqualExpr} \rangle] \gamma \kappa \sigma \\
& | \mathcal{E}[\langle \text{AndExpr}_1 \rangle \& \langle \text{EqualExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{AndExpr}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\
& \quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{EqualExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\
& \quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(r_1 \text{ and}_\perp r_2, \tau, \sigma_2) \text{ where} \\
& \quad \text{if } (\tau_1 == \text{"Z"}) \\
& \quad \quad \tau = \text{"Z"} \\
& \quad \text{else} \\
& \quad \quad \tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2) \\
& \text{endif}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\langle \text{EqualExpr} \rangle] \gamma \kappa \sigma ::= & \\
& \mathcal{E}[\langle \text{RelatExpr} \rangle] \gamma \kappa \sigma \\
& | \mathcal{E}[\langle \text{EqualExpr}_1 \rangle == \langle \text{RelatExpr} \rangle] \gamma \kappa \sigma = \\
& \quad \mathcal{E}[\langle \text{EqualExpr}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\
& \quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{RelatExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\
& \quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \mathbf{boolean}, \sigma_2) \text{ where} \\
& \quad \text{if } (\mathbf{isNumeric}(\tau_1)) \\
& \quad \quad \text{let } (\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)) \text{ and} \\
& \quad \quad \quad r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1)) \text{ and} \\
& \quad \quad \quad r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2)) \text{ in} \\
& \quad \quad \text{if } (\tau = \text{"F"} \text{ or } \tau = \text{"D"}) \\
& \quad \quad \quad \text{if } (r'_1 == \mathbf{NAN} \text{ or } r'_2 == \mathbf{NAN}) \\
& \quad \quad \quad \quad t = \mathbf{false} \\
& \quad \quad \quad \text{else if } (|r'_1| == 0 \text{ and } |r'_2| == 0) \\
& \quad \quad \quad \quad t = \mathbf{true} \\
& \quad \quad \quad \text{else if } (r'_1 == r'_2) \\
& \quad \quad \quad \quad t = \mathbf{true}
\end{aligned}$$

```

        endif
    else if ( $r'_1 == r'_2$ )
         $t = \mathbf{true}$ 
    endif
    else if ( $\tau == \mathbf{"Z"}$ )
         $t = (r_1 == r_2)$ 
    else // must be ref types
    endif
|  $\mathcal{E}[\langle \mathbf{EqualExpr}_1 \rangle \mathbf{!} = \langle \mathbf{RelatExpr} \rangle] \gamma \kappa \sigma =$ 
 $\mathcal{E}[\langle \mathbf{!}(\langle \mathbf{EqualExpr}_1 \rangle == \langle \mathbf{RelatExpr} \rangle)] \gamma \kappa \sigma$ 

 $\mathcal{E}[\langle \mathbf{RelatExpr} \rangle] \gamma \kappa \sigma ::=$ 
 $\mathcal{E}[\langle \mathbf{ShiftExpr} \rangle] \gamma \kappa \sigma$ 
|  $\mathcal{E}[\langle \mathbf{RelatExpr}_1 \rangle < \langle \mathbf{ShiftExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \mathbf{RelatExpr}_1 \rangle] \gamma \kappa_1 \sigma$  where
 $\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \mathbf{ShiftExpr} \rangle] \gamma \kappa_2 \sigma_1$  where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \mathbf{"Z"}, \sigma_2)$  where
    let ( $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$ ) and
         $r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1))$  and
         $r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2))$  in
         $q = r'_1 <_{\tau} r'_2$ 
    similar for  $>$ ,  $<=$  and  $>=$ ,
    with the understanding that positive and negative 0 are equal.
|  $\mathcal{E}[\langle \mathbf{RelatExpr}_1 \rangle \mathbf{instanceof} \langle \mathbf{RefType} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \mathbf{RelatExpr}_1 \rangle] \gamma \kappa_1 \sigma_1$  where
 $\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) =$ 
 $\mathcal{E}[\langle \mathbf{RefType} \rangle] \gamma \kappa_2 \sigma_1$  where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(\gamma.\mathbf{instanceof}(\tau_1, \tau_2), \mathbf{"Z"}, \sigma_2)$ 

```

we compute the result of $r_1 <_{\tau} r_2$ using the following table

Computation of $r_1 <_{\tau} r_2$

		r_1				
		NAN	∞	$-\infty$	$ 0 $	other
r_2	NAN	false	false	false	false	false
	∞	false	false	false	false	false
	$-\infty$	false	true	false	true	true
	$ 0 $	false	true	false	false	$r_1 <_{\tau \perp} r_2$
	other	false	true	false	$r_1 <_{\tau \perp} r_2$	$r_1 <_{\tau \perp} r_2$

where $+_{\tau \perp}$ is normal addition

(using either IEEE 754, or twos complement arithmetic)

32 or 64 bit computation is based on the value of τ

this is a strict extension of normal addition

IEEE underflow or overflow returns an ∞ or a 0 value

twos complement overflow or underflow returns the low order bits of the result

Numeric Expressions. The numeric expressions take numeric operands and produce numeric results. Again, the use of floating point values greatly complicates the specification of operations such as addition and multiplication, and thus are defined in tables to simplify the presentation. We also define subtraction

in terms of addition. There is an oversight in the JLS [1] involving multiplication of infinity values. Consistent with the JDK we define $\infty * \infty == \infty$ and $-\infty * \infty == \infty * -\infty == -\infty$.

$$\begin{aligned} \mathcal{E}[\langle \text{ShiftExpr} \rangle] \gamma \kappa \sigma &::= \\ &\mathcal{E}[\langle \text{AddExpr} \rangle] \gamma \kappa \sigma \\ | \mathcal{E}[\langle \text{ShiftExpr}_1 \rangle \langle \langle \text{AddExpr} \rangle \rangle] \gamma \kappa \sigma &= \mathcal{E}[\langle \text{ShiftExpr}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\ &\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{AddExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\ &\quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau'_1, \sigma_2) \text{ where} \\ &\quad \text{let } \tau'_1 = \mathbf{unaryPromoteType}(\tau_1) \text{ and} \\ &\quad \tau'_2 = \mathbf{unaryPromoteType}(\tau_2) \text{ and} \\ &\quad r'_1 = \mathbf{promote}(\tau'_1, (r_1, \tau_1)) \text{ and} \\ &\quad r'_2 = \mathbf{promote}(\tau'_2, (r_2, \tau_2)) \text{ in} \\ &\quad q = \mathbf{leftShift}((r'_1, \tau'_1), (r'_2, \tau'_2)) \\ &\text{similar for } \gg \text{ and } \gg \gg \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{AddExpr} \rangle] \gamma \kappa \sigma &::= \\ &\mathcal{E}[\langle \text{MultExpr} \rangle] \gamma \kappa \sigma \\ | \mathcal{E}[\langle \text{AddExpr}_1 \rangle + \langle \text{MultExpr} \rangle] \gamma \kappa \sigma &= \mathcal{E}[\langle \text{AddExpr} \rangle] \gamma \kappa_1 \sigma \text{ where} \\ &\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{MultExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\ &\quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2) \text{ where} \\ &\quad \text{if } (\tau_1 == \text{"Ljava.lang.String;"}) \text{ or } \tau_2 == \text{"Ljava.lang.String;"}) \\ &\quad \tau = \text{"Ljava.lang.String;" and} \\ &\quad q = r_1 +_{\tau} r_2 \\ &\text{else} \\ &\quad \tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2) \text{ and} \\ &\quad \text{let } r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1)) \text{ and} \\ &\quad r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2)) \text{ in} \\ &\quad q = (r'_1 +_{\tau} r'_2) \\ &\text{endif} \\ | \mathcal{E}[\langle \text{AddExpr} \rangle - \langle \text{MultExpr} \rangle] \gamma \kappa \sigma &= \mathcal{E}[\langle \text{AddExpr} \rangle] \gamma \kappa_1 \sigma \text{ where} \\ &\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{MultExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\ &\quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2) \text{ where} \\ &\quad \tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2) \text{ and} \\ &\quad \text{let } r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1)) \text{ and} \\ &\quad r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2)) \text{ in} \\ &\quad q = ((r'_1, \tau_1) +_{\tau} (-r'_2, \tau_2)) \end{aligned}$$

where we define $(r_1, \tau_1) +_{\tau} (r_2, \tau_2) =$
 if $(\tau == \text{"Ljava.lang.String;"})$
String(r_1, τ_1) + **String**(r_2, τ_2)
 else
 compute the result using the following table
 endif

Computation of $r_1 +_{\tau} r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	NAN	NAN	NAN	NAN	NAN	NAN	NAN
	∞	NAN	∞	NAN	∞	∞	∞
	$-\infty$	NAN	NAN	$-\infty$	$-\infty$	$-\infty$	$-\infty$
	0	NAN	∞	$-\infty$	0	0	r_1
	-0	NAN	∞	$-\infty$	0	-0	r_1
	other	NAN	∞	$-\infty$	r_2	r_2	$r_1 +_{\tau} r_2$

where $+_{\tau}$ is normal addition

(using either IEEE 754, or twos complement arithmetic)

32 or 64 bit computation is based on the value of τ

this is a strict extension of normal addition

IEEE underflow or overflow returns an ∞ or a 0 value

twos complement overflow or underflow returns the low order bits of the result

$\mathcal{E}[\langle \text{MultExpr} \rangle] \gamma \kappa \sigma ::=$
 $\mathcal{E}[\langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma$
 $| \mathcal{E}[\langle \text{MultExpr}_1 \rangle * \langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{MultExpr}_1 \rangle] \gamma \kappa_1 \sigma$ where
 $\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{UnaryExpr} \rangle] \gamma \kappa_2 \sigma_1$ where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$ where
 $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$ and
 let $r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1))$ and
 $r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2))$ in
 $q = (r'_1 *_{\tau} r'_2)$
 $| \mathcal{E}[\langle \text{MultExpr}_1 \rangle / \langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{MultExpr}_1 \rangle] \gamma \kappa_1 \sigma$ where
 $\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{UnaryExpr} \rangle] \gamma \kappa_2 \sigma_1$ where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$ where
 $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$ and
 let $r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1))$ and
 $r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2))$ in
 if $(| r'_2 | == 0)$
 $\theta(\gamma_1, \sigma_3)$ where
 $\theta = \gamma.[\&throw]$ and
 $\sigma_3, r_3, \tau_3 = \sigma.mkException(ArithmeticException)$ and
 $\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$
 else
 $q = (r'_1 /_{\tau} r'_2)$
 endif
 $| \mathcal{E}[\langle \text{MultExpr}_1 \rangle \% \langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{MultExpr}_1 \rangle] \gamma \kappa_1 \sigma$ where
 $\forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{UnaryExpr} \rangle] \gamma \kappa_2 \sigma_1$ where
 $\forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$ where
 $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$ and
 let $r'_1 = \mathbf{promote}(\tau, (r_1, \tau_1))$ and
 $r'_2 = \mathbf{promote}(\tau, (r_2, \tau_2))$ in
 $q = (r'_1 \%_{\tau} r'_2)$

where we compute $r_1 *_{\tau} r_2$ using the following table

Computation of $r_1 *_{\tau} r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	NAN	NAN	NAN	NAN	NAN	NAN	NAN
	∞	NAN	∞	$-\infty$	NAN	NAN	(s) ∞
	$-\infty$	NAN	$-\infty$	∞	NAN	NAN	(s) ∞
	0	NAN	NAN	NAN	0	-0	(s)0
	-0	NAN	NAN	NAN	-0	0	(s)0
	other	NAN	(s) ∞	(s) ∞	(s)0	(s)0	$r_1 *_{\tau\perp} r_2$

where $*_{\tau\perp}$ is normal multiplication
 (using either IEEE 754, or twos complement arithmetic)
 32 or 64 bit computation is based on the value of τ
 this is a strict extension of normal addition
 IEEE underflow or overflow returns an ∞ or a 0 value
 twos complement overflow or underflow returns the low order bits of the result
 (s) represents the sign of the result which is positive if both
 r_1 and r_2 have the same sign and negative otherwise

where we compute $r_1 /_{\tau} r_2$ using the following table

Computation of $r_1 /_{\tau} r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	NAN	NAN	NAN	NAN	NAN	NAN	NAN
	∞	NAN	NAN	NAN	0	-0	(s)0
	$-\infty$	NAN	NAN	NAN	-0	0	(s)0
	0	NAN	∞	$-\infty$	NAN	NAN	(s) ∞
	-0	NAN	$-\infty$	∞	NAN	NAN	(s) ∞
	other	NAN	(s) ∞	(s) ∞	(s)0	(s)0	$r_1 /_{\tau\perp} r_2$

where $/_{\tau\perp}$ is normal division
 (using either IEEE 754, or twos complement arithmetic)
 32 or 64 bit computation is based on the value of τ
 this is a strict extension of normal addition
 IEEE underflow or overflow returns an ∞ or a 0 value
 twos complement overflow or underflow returns the low order bits of the result
 (s) represents the sign of the result which is positive if both
 r_1 and r_2 have the same sign and negative otherwise

where we compute $r_1 \%_{\tau} r_2$ using the following table

Computation of $r_1 \%_{\tau} r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	NAN	NAN	NAN	NAN	NAN	NAN	NAN
	∞	NAN	NAN	NAN	0	-0	r_1
	$-\infty$	NAN	NAN	NAN	0	-0	r_1
	0	NAN	NAN	NAN	NAN	NAN	NAN
	-0	NAN	NAN	NAN	NAN	NAN	NAN
	other	NAN	NAN	NAN	0	-0	$r_1 \%_{\tau} r_2$

where $\%_{\tau}$ is integer division

(using C/C++ style remainder, or twos complement arithmetic)

Does NOT follow IEEE 754 remainder operation,

rather C/C++ style integer remainder operation

Floating point underflow or overflow returns an ∞ or a 0 value

twos complement overflow or underflow returns the low order bits of the result

(s) represents the sign of the result which is positive if both

r_1 and r_2 have the same sign and negative otherwise

4.14 Location Expressions

All Java expressions return either a value, variable or void (for method invocations that return no value). Unary and primary expressions are the only expressions that can return a variable (location in a store). As such, we use the location semantic function to evaluate unary and primary expressions. However, to maintain consistency in the grammar, we have included regular expression productions and semantics interleaved with the location expressions.

Unary Expressions. Unary expressions involve changing the sign or type of an expression, or incrementing or decrementing a value. In the case of pre or post increment or decrement operations the expression has a definite side-effect on the store, as is indicated in the semantics. Note that the return value of the expression indicates the pre or post nature of the expression. If the unary (or primary) expression does not return a variable, then the value *undef* is returned.

$$\mathcal{E}[\llbracket \langle \text{UnaryExpr} \rangle \rrbracket \gamma \kappa \sigma] = \mathcal{L}[\llbracket \langle \text{UnaryExpr} \rangle \rrbracket \gamma \alpha \sigma] \text{ where } \forall r, \tau, l, \sigma_1. \alpha(r, \tau, l, \sigma_1) = \kappa(r, \tau, \sigma_1)$$

$$\begin{aligned} \mathcal{L}[\llbracket \langle \text{UnaryExp} \rangle \rrbracket \gamma \alpha \sigma] ::= & \\ & \mathcal{L}[\llbracket \langle \text{PreIncExpr} \rangle \rrbracket \gamma \kappa \sigma] \\ & | \mathcal{L}[\llbracket \langle \text{PreDecExpr} \rangle \rrbracket \gamma \kappa \sigma] \\ & | \mathcal{L}[\llbracket \langle \text{UnaryExprNotPlusMinus} \rangle \rrbracket \gamma \kappa \sigma] \\ & | \mathcal{L}[\llbracket + \langle \text{UnaryExpr}_1 \rangle \rrbracket \gamma \alpha \sigma] = \mathcal{L}[\llbracket \langle \text{UnaryExpr}_1 \rangle \rrbracket \gamma \alpha_1 \sigma] \text{ where } \\ & \quad \forall r, \tau, l, \sigma_1. \alpha_1(r, \tau, l, \sigma_1) = \alpha(r, \tau, \text{undef}, \sigma_1) \\ & | \mathcal{L}[\llbracket - \langle \text{UnaryExpr}_1 \rangle \rrbracket \gamma \alpha \sigma] = \mathcal{L}[\llbracket \langle \text{UnaryExpr}_1 \rangle \rrbracket \gamma \alpha_1 \sigma] \text{ where } \\ & \quad \forall r, \tau, l, \sigma_1. \alpha_1(r, \tau, l, \sigma_1) = \alpha(0 \text{ } -_{\tau} r, \tau, \text{undef}, \sigma_1) \end{aligned}$$

$$\begin{aligned} \mathcal{L}[\llbracket \langle \text{UnaryExprNotPlusMinus} \rangle \rrbracket \gamma \kappa \sigma] ::= & \\ & \mathcal{L}[\llbracket \langle \text{PostExpr} \rangle \rrbracket \gamma \kappa \sigma] \end{aligned}$$

$$\begin{array}{l}
| \mathcal{L}[\langle \text{CastExpr} \rangle] \gamma \kappa \sigma \\
| \mathcal{E}[\sim \langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma = \mathcal{L}[\langle \text{UnaryExpr} \rangle] \gamma \alpha \sigma \text{ where} \\
\quad \forall r, \tau, l, \sigma_1. \alpha(r, \tau, l, \sigma_1) = \\
\quad \quad \text{let } \tau_1 = \mathbf{unaryPromoteType}(\tau) \text{ and} \\
\quad \quad \quad r_1 = \mathbf{promote}(\tau_1, (r, \tau)) \text{ in} \\
\quad \quad \quad \kappa((-r_1) - 1, \tau, \sigma_1) \\
| \mathcal{E}[\! \langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma = \mathcal{L}[\langle \text{UnaryExpr} \rangle] \gamma \alpha \sigma \text{ where} \\
\quad \forall r, \tau, l, \sigma_1. \alpha_1(r, \tau, l, \sigma_1) = \\
\quad \quad \text{if } (r == \mathbf{true}) \\
\quad \quad \quad \kappa(\mathbf{false}, \tau, \sigma_2) \\
\quad \quad \text{else} \\
\quad \quad \quad \kappa(\mathbf{false}, \tau, \sigma_2) \\
\quad \text{endif}
\end{array}$$

We have reverted to the non-LALR(1) grammar for cast expressions to simplify the presentation of the semantics. Specifically, the return type of the expression is the type of the cast (given that no error occurs), and the return value is the converted value of the expression.

$$\begin{array}{l}
\mathcal{E}[\langle \text{CastExpr} \rangle] \gamma \kappa \sigma ::= \\
\quad \mathcal{E}[(\langle \text{PrimType} \rangle \langle \text{Dims} \rangle^?) \langle \text{UnaryExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{UnaryExpr} \rangle] \gamma \kappa_1 \sigma \text{ where} \\
\quad \quad \text{let } \tau' = \mathcal{T}[\langle \text{PrimType} \rangle] \gamma \text{ and} \\
\quad \quad \quad d = \mathbf{fst}(\mathcal{V}[\langle \text{Dims} \rangle] \gamma) \text{ and} \\
\quad \quad \quad \tau = \mathbf{mkArrayType}(\tau', d) \text{ in} \\
\quad \quad \forall r, \tau_1, \sigma. \kappa_1(r, \tau_1, \sigma) = \kappa(r_1, \tau, \sigma) \text{ where} \\
\quad \quad \quad r_1 = \mathbf{cast}(\tau, (r, \tau_1)) \\
| \mathcal{E}[(\langle \text{RefType} \rangle) \langle \text{UnaryExprNotPlusMinus} \rangle] \gamma \kappa \sigma = \\
\quad \mathcal{E}[\langle \text{UnaryExprNotPlusMinus} \rangle] \gamma \kappa_1 \sigma \text{ where} \\
\quad \quad \text{let } \tau = \mathcal{T}[\langle \text{RefType} \rangle] \gamma \text{ in} \\
\quad \quad \forall r, \tau_1, \sigma. \kappa_1(r, \tau_1, \sigma) = \\
\quad \quad \quad \text{if } (\text{not } (\mathit{env.assnCompatible}(\tau, \tau_1) \text{ or} \\
\quad \quad \quad \mathit{env.assnCompatible}(\tau_1, \tau))) \\
\quad \quad \quad \theta(\gamma_1, \sigma_2) \text{ where} \\
\quad \quad \quad \quad \theta = \gamma.[\&throw] \text{ and} \\
\quad \quad \quad \quad \sigma_2, r_2, \tau_2 = \sigma.\mathit{mkException}(\mathit{CastConversionException}) \text{ and} \\
\quad \quad \quad \quad \gamma_1 = \gamma[\&thrown \leftarrow (r_2, \tau_2)] \\
\quad \quad \text{else} \\
\quad \quad \quad \kappa(r_1, \tau, \sigma) \text{ where} \\
\quad \quad \quad \quad r_1 = \mathbf{cast}(\tau, (r, \tau_1)) \\
\quad \text{endif}
\end{array}$$

In the JLS [1], there is discussion that $(p)++$ is a valid post fix operation (“ $(p)++$ can make sense only as a postfix increment of p ”). However, in the JDK, any parenthesized expression returns only a value and not a variable. We follow that convention here.

$$\begin{array}{l}
\mathcal{L}[\langle \text{PostIncExpr} \rangle] \gamma \alpha \sigma ::= \\
\quad \mathcal{L}[\langle \text{PostExpr} \rangle ++] \gamma \alpha \sigma = \mathcal{L}[\langle \text{PostExpr} \rangle] \gamma \alpha_1 \sigma \text{ where} \\
\quad \quad \forall r, \tau_1, l, \sigma_1. \alpha_1(r, \tau_1, l, \sigma_1) = \\
\quad \quad \quad \text{let } \tau = \mathbf{binaryPromotionType}(\tau, \text{"I"}) \text{ and}
\end{array}$$

$$\begin{aligned}
& r_1 = \mathbf{promote}(\tau, (r, \tau_1)) \text{ and} \\
& r_2 = \mathbf{promote}(\tau, (1, \text{"I''})) \text{ and} \\
& q = \mathbf{cast}(\tau_1, (r_1 +_{\tau_{\perp}} r_2), \tau) \text{ in} \\
& \alpha(r, \tau_1, \mathit{undef}, \sigma_1[l \leftarrow q])
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}[\langle \text{PostDecExpr} \rangle] \gamma \alpha \sigma & ::= \\
& \mathcal{L}[\langle \text{PostExpr} \rangle - -] \gamma \alpha \sigma = \mathcal{L}[\langle \text{PostExpr} \rangle] \gamma \alpha_1 \sigma \text{ where} \\
& \forall l, r, \tau_1, l, \sigma_1. \alpha_1(r, \tau_1, l, \sigma_1) = \\
& \text{let } \tau = \mathbf{binaryPromotionType}(\tau, \text{"I''}) \text{ and} \\
& r_1 = \mathbf{promote}(\tau, (r, \tau_1)) \text{ and} \\
& r_2 = \mathbf{promote}(\tau, (-1, \text{"I''})) \text{ and} \\
& q = \mathbf{cast}(\tau_1, (r_1 +_{\tau_{\perp}} r_2), \tau) \text{ in} \\
& \alpha(r, \tau_1, \mathit{undef}, \sigma_1[l \leftarrow q])
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}[\langle \text{PostExpr} \rangle] \gamma \alpha \sigma & ::= \\
& \mathcal{L}[\langle \text{Primary} \rangle] \gamma \alpha \sigma \\
& | \mathcal{L}[\langle \text{Name} \rangle] \gamma \alpha \sigma \\
& | \mathcal{L}[\langle \text{PostIncExpr} \rangle] \gamma \alpha \sigma \\
& | \mathcal{L}[\langle \text{PostDecExpr} \rangle] \gamma \alpha \sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}[\langle \text{PreIncExpr} \rangle] \gamma \alpha \sigma & ::= \\
& \mathcal{L}[++ \langle \text{UnaryExpr} \rangle] \gamma \alpha \sigma = \mathcal{L}[\langle \text{UnaryExpr} \rangle] \gamma \alpha_1 \sigma \text{ where} \\
& \forall r, \tau_1, l, \sigma_1. \alpha_1(r, \tau_1, l, \sigma_1) = \\
& \text{let } \tau = \mathbf{binaryPromotionType}(\tau, \text{"I''}) \text{ and} \\
& r_1 = \mathbf{promote}(\tau, (r, \tau_1)) \text{ and} \\
& r_2 = \mathbf{promote}(\tau, (1, \text{"I''})) \text{ and} \\
& q = \mathbf{cast}(\tau_1, (r_1 +_{\tau_{\perp}} r_2), \tau) \text{ in} \\
& \alpha(q, \tau_1, \mathit{undef}, \sigma_1[l \leftarrow q])
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}[\langle \text{PreDecExpr} \rangle] \gamma \alpha \sigma & ::= \\
& \mathcal{L}[- - \langle \text{UnaryExpr} \rangle] \gamma \alpha \sigma = \mathcal{L}[\langle \text{UnaryExpr} \rangle] \gamma \alpha_1 \sigma \text{ where} \\
& \forall r, \tau_1, l, \sigma_1. \alpha_1(r, \tau_1, l, \sigma_1) = \\
& \text{let } \tau = \mathbf{binaryPromotionType}(\tau, \text{"I''}) \text{ and} \\
& r_1 = \mathbf{promote}(\tau, (r, \tau_1)) \text{ and} \\
& r_2 = \mathbf{promote}(\tau, (-1, \text{"I''})) \text{ and} \\
& q = \mathbf{cast}(\tau_1, (r_1 +_{\tau_{\perp}} r_2), \tau) \text{ in} \\
& \alpha(q, \tau_1, \mathit{undef}, \sigma_1[l \leftarrow q])
\end{aligned}$$

Primary Expressions. These expressions are the base expressions of the Java language providing access to variables, fields, methods, arrays and new object instances.

$$\begin{aligned}
\mathcal{L}[\langle \text{Primary} \rangle] \gamma \alpha \sigma & ::= \\
& \mathcal{L}[\langle \text{PrimaryNoNewArray} \rangle] \gamma \alpha \sigma \\
& | \mathcal{L}[\langle \text{ArrayCreationExpr} \rangle] \gamma \alpha \sigma = \mathcal{E}[\langle \text{ArrayCreationExpr} \rangle] \gamma \kappa \sigma \text{ where} \\
& \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) = \alpha(r, \tau, \mathit{undef}, \sigma_1)
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}[\langle \text{PrimaryNoNewArray} \rangle] \gamma \alpha \sigma & ::= \\
& \mathcal{L}[\langle \text{Literal} \rangle] \gamma \alpha \sigma = \\
& \text{let } (r, \tau) = \mathcal{V}[\langle \text{Literal} \rangle] \gamma \text{ in} \\
& \alpha(r, \tau, \mathit{undef}, \sigma)
\end{aligned}$$

$$\begin{aligned}
& | \mathcal{L}[\mathbf{this}] \gamma \alpha \sigma = \alpha(\gamma[\&thisObject], \gamma[\&thisClass], undef, \sigma) \\
& | \mathcal{L}[(\langle \text{Expr} \rangle)] \gamma \alpha \sigma = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\
& \quad \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) = \alpha(r, \tau, undef, \sigma_1) \\
& | \mathcal{L}[\langle \text{ClassInstCreationExpr} \rangle] \gamma \alpha \sigma \\
& | \mathcal{L}[\langle \text{FieldAccess} \rangle] \gamma \alpha \sigma \\
& | \mathcal{L}[\langle \text{MethodInv} \rangle] \gamma \alpha \sigma = \mathcal{E}[\langle \text{MethodInv} \rangle] \gamma \kappa \sigma \text{ where} \\
& \quad \forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) = \alpha(r, \tau, undef, \sigma_1) \\
& | \mathcal{L}[\langle \text{ArrayAccess} \rangle] \gamma \alpha \sigma
\end{aligned}$$

Array creation expressions are responsible for the creation of a new array of values. Specifically, they allocate space in the store for the array, and then initialize all of the elements of the array based on the default initializer for the array elements. Note that if these elements are reference types, they are initialized to `null`. This expression only returns a value, the reference to the array, and not a location.

$$\begin{aligned}
& \mathcal{E}[\langle \text{ArrayCreationExpr} \rangle] \gamma \kappa \sigma ::= \\
& \quad \mathcal{E}[\mathbf{new} \langle \text{PrimType} \rangle \langle \text{DimExprList} \rangle \langle \text{Dims} \rangle^?] \gamma \kappa_1 \sigma = \\
& \quad \mathcal{E}[\langle \text{DimExprList} \rangle] \gamma \kappa \sigma \text{ where} \\
& \quad \forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) = \kappa(q, \tau_1, \sigma_2) \text{ where} \\
& \quad \quad v = \text{fst}(\mathcal{V}[\langle \text{Dims} \rangle] \gamma) \text{ and} \\
& \quad \quad \tau_p = \mathcal{T}[\langle \text{PrimType} \rangle] \gamma \text{ and} \\
& \quad \quad \tau_1 = \mathbf{mkArrayType}(\tau, v) \text{ and} \\
& \quad \quad (\sigma_2, q) = \sigma_1.\text{allocateArray}(\tau_1, \tau_p) \\
& | \mathcal{E}[\mathbf{new} \langle \text{ClassInterfaceType} \rangle \langle \text{DimExprList} \rangle \langle \text{Dims} \rangle^?] \gamma \kappa \sigma = \\
& \quad \mathcal{E}[\langle \text{DimExprList} \rangle] \gamma \kappa_1 \sigma_1 \text{ where} \\
& \quad \quad \text{sto}_1 = \gamma.\text{classLoader}(\mathbf{fst}(\mathcal{V}[\langle \text{ClassInterfaceType} \rangle] \gamma), \sigma) \text{ and} \\
& \quad \quad \forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) = \kappa(q, \tau_1, \sigma_2) \text{ where} \\
& \quad \quad \quad v = \text{fst}(\mathcal{V}[\langle \text{Dims} \rangle] \gamma) \text{ and} \\
& \quad \quad \quad \tau_p = \mathcal{T}[\langle \text{ClassInterfaceType} \rangle] \gamma \text{ and} \\
& \quad \quad \quad \tau_1 = \mathbf{mkArrayType}(\tau, v) \text{ and} \\
& \quad \quad \quad (\sigma_2, q) = \sigma_1.\text{allocateArray}(\tau_1, \tau_p)
\end{aligned}$$

The following semantics denote field access. Specifically, these semantics look up the named field in the environment and return the fields location and type.

$$\begin{aligned}
& \mathcal{L}[\langle \text{FieldAccess} \rangle] \gamma \alpha \sigma ::= \\
& \quad \mathcal{L}[\langle \text{Primary} \rangle . \langle \text{Id} \rangle] \gamma \alpha \sigma = \alpha(r, \tau, l, \sigma) \text{ where} \\
& \quad \quad r = \sigma[l] \text{ and} \\
& \quad \quad l, \tau = \gamma[\mathcal{V}[\langle \text{Primary} \rangle . \langle \text{Id} \rangle] \gamma] \\
& | \mathcal{L}[\mathbf{super} . \langle \text{Id} \rangle] \gamma \alpha \sigma = \alpha(r, \tau, l, \sigma) \text{ where} \\
& \quad \quad r = \sigma[l] \text{ and} \\
& \quad \quad l, \tau = \gamma[\mathcal{V}[\gamma[\&super] . \langle \text{Id} \rangle] \gamma]
\end{aligned}$$

The following semantics denote the process of invoking a method call. We have simplified the syntax here from the JLS by just specifying a Name for the method instead of separating the primary and super constructs. The concept for this access is detailed in the auxiliary functions that search the environment. The result of the environment search and retrieval is a function that takes an environment, command continuation and a store and returns an answer. These semantics evaluate the arguments, look up the function for the specified method and execute that function.

$$\begin{aligned}
\mathcal{E}[\langle \text{MethodInv} \rangle] \gamma \kappa \sigma &::= \\
\mathcal{E}[\langle \text{Name} \rangle (\langle \text{ArgList} \rangle^?)] \gamma \kappa \sigma &= \\
\mathcal{E}[\langle \text{ArgList} \rangle] \gamma \kappa_1 \sigma &\text{ where} \\
\forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma_1) &= m(\gamma \theta \sigma_1) \text{ where} \\
\text{sig} &= \mathbf{getSigs}(r) \text{ and} \\
m &= \gamma.\mathit{getMethod}(\mathbf{fst} \mathcal{V}[\langle \text{Name} \rangle] \gamma, \text{sig}) \text{ and} \\
\forall \gamma_2, \sigma_2. \theta(\gamma_2, \sigma_2) &= \kappa(\gamma_2[\&returnVal], \gamma_2[\&returnType], \sigma_2)
\end{aligned}$$

The following semantics are used to specify the creation of an instance of a class through the invocation of a `new` operator. Upon invocation of this operator the class needs to be loaded (if it had not been loaded). Loading involves creation of storage space in the store, execution of field initializers for static fields of the class, and execution of static constructors for the class. After the execution of these entities, only then is the explicit constructor invoked (and its arguments evaluated). Note the inclusion of the `<ClassBody>` construct in the last two productions. These are new as of Java 1.1 and permit the construction of anonymous classes. For the sake of brevity, we do not include their semantics here. These semantics would be the same as the first semantics except that the execution method m would be the evaluation of the class body.

$$\begin{aligned}
\mathcal{E}[\langle \text{ClassInstCreationExpr} \rangle] \gamma \kappa \sigma &::= \\
\mathcal{E}[\mathbf{new} \langle \text{ClassType} \rangle (\langle \text{ArgList} \rangle^?)] \gamma \kappa \sigma &= \\
\mathcal{E}[\langle \text{ArgList} \rangle] \gamma \kappa_1 \sigma_1 &\text{ where} \\
\text{sto}_1 &= \gamma.\mathit{classLoader}(\mathbf{fst}(\mathcal{V}[\langle \text{ClassType} \rangle]), \sigma) \text{ and} \\
\forall r, \tau, \sigma_2. \kappa_1(r, \tau, \sigma_2) &= m(\gamma \theta \sigma_2) \text{ where} \\
\text{sig} &= \mathbf{getSigs}(r) \text{ and} \\
m &= \gamma.\mathit{getMethod}(\mathbf{fst} \mathcal{V}[\langle \text{ClassType} \rangle] \gamma, \text{sig}) \text{ and} \\
\forall \gamma_2, \sigma_2. \theta(\gamma_2, \sigma_2) &= \kappa(\gamma_2[\&returnVal], \gamma_2[\&returnType], \sigma_2) \\
| \mathbf{new} \langle \text{ClassType} \rangle (\langle \text{ArgList} \rangle^?) &\langle \text{ClassBody} \rangle \\
| \mathbf{new} \langle \text{InterfaceType} \rangle () &\langle \text{ClassBody} \rangle
\end{aligned}$$

The array access expressions allow us to dereference an existing array and return the location of an element of the array. If that element is a reference type, then the location returned is the location that stores the references and not the location of the object itself.

$$\begin{aligned}
\mathcal{L}[\langle \text{Array Access} \rangle] \gamma \alpha \sigma &::= \\
\mathcal{L}[\langle \text{Name} \rangle [\langle \text{Expr} \rangle]] \gamma \alpha_1 \sigma &= \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma \text{ where} \\
\forall r, \tau, \sigma_1. \kappa(r, \tau, \sigma_1) &= \alpha(q, \tau_1, l, \sigma_1) \text{ where} \\
v &= \mathbf{fst}(\mathcal{V}[\langle \text{Name} \rangle] \gamma) \text{ and} \\
a &= \gamma.\mathit{getArrayRef}(v) \text{ and} \\
\tau' &= \mathbf{unaryPromoteType}(\tau) \text{ and} \\
l &= \gamma.\mathit{getArrayElem}(a, \mathbf{promote}(\tau', (r, \tau))) \text{ and} \\
\tau_1 &= \gamma.\mathit{getArrayElemType}(a) \text{ and} \\
q &= \sigma(l) \\
| \mathcal{L}[\langle \text{PrimaryNoNew Array} \rangle [\langle \text{Expr} \rangle]] \gamma \alpha \sigma &= \\
\mathcal{L}[\langle \text{PrimaryNoNew Array} \rangle] \gamma \alpha_1 \sigma &\text{ where} \\
\forall r_1, \tau_1, l_1, \sigma_1. \alpha r, \tau, l, \sigma_1 &= \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma_1 \text{ where} \\
\forall r_2, \tau_2, \sigma_2. \kappa r_2, \tau_2, \sigma_2 &= \alpha(q, \tau_1, l, \sigma_2) \text{ where} \\
\tau' &= \mathbf{unaryPromoteType}(\tau_2) \text{ and} \\
l &= \gamma.\mathit{getArrayElem}(r_1, \mathbf{promote}(\tau', (r_2, \tau_2))) \text{ and}
\end{aligned}$$

$$\begin{aligned} \tau_1 &= \gamma.getArrayElemType(r_1) \text{ and} \\ q &= \sigma(l) \end{aligned}$$

The `<ArgList>` construction allows us to specify a list of expressions. The result is a list of pairs of values and types for each of the arguments in the argument list.

$$\begin{aligned} \mathcal{E}[\langle \text{ArgList} \rangle] \gamma \kappa \sigma &::= \\ &[\mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma] \\ &| \mathcal{E}[\langle \text{ArgList}_1 \rangle, \langle \text{Expr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{ArgList}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\ &\quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\ &\quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2) \text{ where} \\ &\quad q = \mathbf{append}(r_1, r_2) \text{ and} \\ &\quad \tau = \tau_1 + \tau_2 \end{aligned}$$

Dims. The following three productions are used in cast expressions and array creation expressions to specify the dimensions of the created array. The information obtained from these productions is used to provide a count of the number of array indices and any specified dimension sizes.

$$\begin{aligned} \mathcal{E}[\langle \text{DimExprList} \rangle] \gamma \kappa \sigma &::= \\ &\mathcal{E}[\langle \text{DimExpr} \rangle] \gamma \kappa \sigma \\ &| \mathcal{E}[\langle \text{DimExprList}_1 \rangle \langle \text{DimExpr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{DimExprList}_1 \rangle] \gamma \kappa_1 \sigma \text{ where} \\ &\quad \forall r_1, \tau_1, \sigma_1. \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\langle \text{DimExpr} \rangle] \gamma \kappa_2 \sigma_1 \text{ where} \\ &\quad \forall r_2, \tau_2, \sigma_2. \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2) \text{ where} \\ &\quad q = \mathbf{append}(r_1, r_2) \text{ and} \\ &\quad \tau = \mathbf{mkArrayType}(\tau_1, 1) \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{DimExpr} \rangle] \gamma \kappa \sigma &::= \\ &\mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa \sigma = \mathcal{E}[\langle \text{Expr} \rangle] \gamma \kappa_1 \sigma \text{ where} \\ &\quad \forall r, \tau, \sigma_1. \kappa_1(r, \tau, \sigma) = \kappa(q, \mathbf{int}[], \sigma_1) \text{ where} \\ &\quad \tau' = \mathbf{unaryPromoteType}(\tau) \text{ and} \\ &\quad q = [\mathbf{promote}(\tau', (r, \tau))] \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\langle \text{Dims} \rangle] \gamma \kappa \sigma &::= \\ &\mathcal{V}[\langle \rangle] \gamma = (1, \mathbf{int}) \\ &| \mathcal{V}[\langle \text{Dims}_1 \rangle \langle \rangle] \gamma = (v + 1, \mathbf{int}) \text{ where} \\ &\quad (v, \tau) = \mathcal{V}[\langle \text{Dims}_1 \rangle] \gamma \end{aligned}$$

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
2. IEEE. IEEE standard for binary floating-point arithmetic, 1985.
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
4. J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. M.I.T. Press, 1977.
5. R.D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.