K-Java: Runtime Semantics for Method Invocation and Object Instantiation

Denis Bogdănaş

Alexandru Ioan Cuza University, Iaşi, Romania
denis.bogdanas@info.uaic.ro

October 26, 2014

1 Method Invocation

1.1 Background

In this section we present the fragment of configuration used by runtime method invocation. The figure below contains the cells and their sorts.

$$\left\langle \left\langle \left\langle \langle K\rangle_{\mathsf{k}}\ \langle \mathit{List}[\,(K, \mathit{Bag})\,]\rangle_{\mathsf{stack}}\ \langle \langle \mathit{Map}[\mathit{Id} \mapsto \mathit{Int}]\rangle_{\mathsf{env}}\ \langle \mathit{ClassType}\rangle_{\mathsf{crntClass}}\ \langle \mathit{Int}\rangle_{\mathsf{location}}\rangle_{\mathsf{methodContext}} \right\rangle_{\mathsf{thread*}} \right\rangle_{\mathsf{threads}} \left\langle \left\langle \begin{matrix} \langle \mathit{ClassType}\rangle_{\mathsf{classType}}\ \langle \mathit{ClassMetaType}\rangle_{\mathsf{classMetaType}} \\ \langle \mathit{Map}[\mathit{Signature} \mapsto \mathit{ClassType}]\rangle_{\mathsf{methods}} \\ \left\langle \left\langle \begin{matrix} \langle \mathit{Signature}\rangle_{\mathsf{methodSignature}} \\ \langle \mathit{List}[\,\mathit{MethodParam}\,]\rangle_{\mathsf{methodParams}} \\ \langle K\rangle_{\mathsf{methodBody}} \\ \langle \mathit{AccessMode}\rangle_{\mathsf{methodAccessMode}} \\ \langle \mathit{ContextType}\rangle_{\mathsf{methodContextType}} \end{matrix} \right\rangle_{\mathsf{methodDec*}} \right\rangle_{\mathsf{methodDecs}} \end{matrix} \right\rangle_{\mathsf{class*}} \right\rangle_{\mathsf{classes}} \right\rangle_{\top}$$

The cell $\langle\rangle_{\mathsf{k}}$ stores the current computation. The cell $\langle\rangle_{\mathsf{stack}}$ is a list of pairs of the form (K, Bag), and represents the standard method call stack. The first element represents the remaining computation at the moment the method was called. The second element of sort Bag represents the content of cell $\langle\rangle_{\mathsf{methodContext}}$ at the moment of method call.

The cell $\langle\rangle_{\mathsf{class}}$ contains various sub-cells holding the content of that class. The first cell in $\langle\rangle_{\mathsf{classType}}$ of sort ClassType that holds the fully qualified class name. This cell serves as a key in all rules that match a fragment of a $\langle\rangle_{\mathsf{class}}$. The value in the cell $\langle\rangle_{\mathsf{classMetaType}}$ is either "class" or "interface". From now on we will refer to both meta types as classes, referring to metatype value when distinction is necessary. The next cell is $\langle\rangle_{\mathsf{methods}}$. This is a map from method signatures to classes where the respective signatures are declared. It contains not only the methods declared in this class, but also those inherited from the base class, but not from the base interfaces. By "inherited" here we mean all the methods contained in the cell $\langle\rangle_{\mathsf{methods}}$ of the base class that were not overridden by a method declaration with the same signature in the current class. This definition is different from the inheritance rules in JLS §8.4.6, although the difference is only relevant at the elaboration time.

The cell $\langle\rangle_{\mathsf{methodDec}}$ represents a method declared inside the current class. The subcell $\langle\rangle_{\mathsf{methodSignature}}$ is the key for accessing other cells for this declaration. The other cells are the parameters, the body, the access mode (private, public etc.) and the context type (either instance or static).

In order for strictness and context rules to work we have to define some K productions as KResult. The most common forms of KResult in Java are the following:

$$\text{SYNTAX} \quad \mathit{KResult} ::= \mathit{ClassType} \\ \quad\quad\quad\quad \mid \mathit{TypedVal}$$

The first represents a class. Second is a typed value, the result of evaluation of any expression. The forms of typed values relevant for method invocation are object reference and null:

$$\text{SYNTAX} \quad \mathit{TypedVal} ::= \texttt{objectRef}\ (\mathit{Int}, \mathit{ClassType})\ \texttt{::}\ \mathit{ClassType} \\ \quad\quad\quad\quad \mid \texttt{null}\ \texttt{::}\ \mathit{ClassType}$$

The type after four dots (::) separator is the static type associated with that value. The values inside objectRef() are the address inside the store and the runtime type of the object.

For the sake of simplicity we we will also consider $\cdot_K$ - the unit element of K to be KResult. The value $\cdot_K$ is often used in auxiliary functions as a placeholder until some actual value is computed.

1.2 Introduction

An elaborated method invocation expression may have one of the following forms:

- An invocation of a static method qualified by its class: Class.f(args)
- An invocation of a static method qualified by an expression producing an object: o.f(args). Even if the method is static we cannot simply replace the qualifier with its compile-time type at elaboration phase, because the qualifier expression still has to be evaluated and might produce side effects. We cannot replace it with o; class.f(args); either, because o; might be invalid. Not all expressions valid as qualifiers are valid as expression statements (JLS §14.8). We wanted the elaboration result to be a valid Java program, thus we could not afford such a transformation.
- An invocation of an instance method qualified by a class reference: o.f(args)
- An invocation of an instance method qualified by an interface reference: i.f(args)

The evaluation of the method invocation expression consists from 5 steps outlined below. Those steps, unless otherwise specified, are common to all the method call forms enumerated above.

1. Evaluation of the qualifier expression
2. Evaluation of method arguments
3. Computation of static method information
4. Locating the actual method declaration to be invoked
5. Actual method invocation.

In JLS runtime semantics of method invocation is described in §15.2.4. Although there is some correspondence between the steps in our semantics and the steps in JLS, it is generally not one-to-one. JLS description of method invocation consists of the following 5 steps. For each step we give the relevant chapter of the JLS and the step in our semantics.

1. Compute the target reference (§15.12.4.1), semantics step 1
2. Evaluate arguments (§15.12.4.2), semantics step 1
3. Check the accessibility of the method to be invoked (§15.12.4.3), no semantics
4. Locate the actual method code to invoke (§15.12.4.4), semantics step 3
5. Actual method invocation, semantics step 4.

Generally the rules of K-Java do not follow directly the wording of JLS. The reasons for this will choice will be given at the end of the section. The details related to each step are described in the semantics below, above each rule and auxiliary construct. For each rule we will also refer to the respective JLS page, if there is a correspondence.

1.3 Evaluation of the qualifier and the arguments

The first two parts of method invocation logic are evaluation of the qualifier expression and evaluation of the arguments. JLS enforces the following conditions on the order of subexpressions evaluation: - Arguments have to be evaluated after the qualifier was evaluated. This is ensured by checking that the qualifier is of sort KResult at the moment when arguments are heated. - Arguments are evaluated left to right. To ensure this we add a side condition that checks that all the arguments before the one being heated (if any) are already of the sort KResult.

While there are two sections dedicated to this logic (§15.12.4.1, §15.12.4.2), we don't need any K rules for it. Instead, subexpressions evaluation ensured by strictness annotations that accompany the following syntax definitions:

SYNTAX $\mathit{Exp} ::= K\ .\ \mathit{MethodName}(\mathit{Exps})$ [seqstrict(1,3)]

SYNTAX $\mathit{MethodName} ::= \mathit{Id}$

SYNTAX $\mathit{Exps} ::= \mathit{List}\{\mathit{Exp}, \text{“,”}\}$ [seqstrict]

The annotation seqstrict(1,3) on the first definition ensures that arguments are evaluated after the qualifier is evaluated. The qualifier term might be either an expression or a Class. If it is expression, it will be heated and evaluated. If it is a class (for certain static methods), then it is already a KResult strictness rule will have no effect on it. Note that arguments have to be evaluated even in the case when the qualifier evaluated to null. At the same time that if evaluation of the qualifier or any of the arguments completes abruptly, the whole method invocation expression completes abruptly for the same reason. K-Java does not need any special rules to cover those cases. The semantics has a fixed number of rules for throw statement that ensure the correct propagation of exceptions from any context.

1.4 Loading method information

During the second step of the method invocation the second argument of the production is replaced with the auxiliary data structure methodInfo(). This data structure contains the information required to choose the right method lookup strategy at the next step. The production methodInfo() contains the following arguments:

- Method signature Sig
- Qualifying class QualC of the method invocation, e.g. the compile-time type of the qualifier.
- The meta type of QualC - MetaT. It may have one of the two values - class or interface.
- DecC - declaring class, the class where the method was actually declared, as observed by QualT. E.g. the most derived class in QualC hierarchy where there is a declaration of a method with signature Sig.
- ContextT - the context type of the method. Either static, for static methods, or instance for non-static methods.
- Acc - access modifier (private, package, protected or public). For the purpose of uniformity we use the modifier package when no access modifier is provided.

All the information stored in methodInfo() is static. In K-Java we already have this information computed, but it is stored in various cells inside $\langle\rangle_{\mathsf{class}}$ and $\langle\rangle_{\mathsf{classDec}}$. The rules from step 3 simply load the relevant information from configuration cells to methodInfo() arguments.

$$\textsc{syntax}\quad \mathit{MethodName} ::= \texttt{methodInfo}\ (\mathit{Signature}, \mathit{RefType}, \mathit{ClassMetaType}, \mathit{RefType}, \mathit{ContextType}, \mathit{AccessMode})$$

The first rule from this step rewrites the method name into a methodInfo() term whose first argument is the method signature. The auxiliary function getTypes() computes the list of types from the list of parameter declarations. The second argument of methodInfo() is also computed at this step - it is the type of the qualifier. The rest of the arguments are filled in with default values. They will be rewritten into actual values by the following rules.

RULE INVOKE-COMPUTE-METHODINFO-SIGNATURE

$$\mathit{Qual}\ \texttt{.}\ \frac{\mathit{Name}}{\texttt{methodInfo}\ (\ \texttt{sig}\ (\mathit{Name},\ \texttt{getTypes}\ (\mathit{Args})),\ \texttt{typeOf}\ (\mathit{Qual}), \cdot_K, \cdot_K, \cdot_K, \cdot_K)}(\mathit{Args} : \mathit{TypedVals})$$

Note that in this rule variable Args is defined of type TypedVals instead of Exps. This restriction ensures that arguments (and consequently the qualifier) are already evaluated at the moment when this rule is invoked. The sort TypedVals represents a list of typed values, the evaluation result of Exps. It is defined as following:

$$\textsc{syntax}\quad \mathit{TypedVals} ::= \mathit{List}\{\mathit{TypedVal}, \text{“,”}\}$$

$$\textsc{syntax}\quad \mathit{Exps} ::= \mathit{TypedVals}$$

Because TypedVal is subsorted to KResult, TypedVals being a list of KResult is implicitly subsorted to KResult.

The second rule for method invocation loads MetaT and DecC. It requires Sig and QualC computed by the previous rule.

RULE INVOKE-COMPUTE-METHODINFO-DECC

$$\left\langle \_\ \texttt{.}\ \texttt{methodInfo}\ (\mathit{Sig}, \mathit{QualC}, \frac{\cdot_K}{\mathit{MetaT}}, \frac{\cdot_K}{\mathit{DecC}}, \_, \_)(\_)\ \cdots \right\rangle_{\mathsf{k}}$$
$$\langle \mathit{QualC} \rangle_{\mathsf{classType}}\ \ \langle \mathit{MetaT} \rangle_{\mathsf{classMetaType}}\ \ \langle \cdots\ \mathit{Sig} \mapsto \mathit{DecC}\ \cdots \rangle_{\mathsf{methods}}$$

There is one case that is not covered by the previous rule - the case when the cell $\langle\rangle_{\mathsf{methods}}$ does not have a key equal to Sig. This is possible in one of the following situations:

- Qualifying type is an interface.
- Qualifying type is an abstract class. The called method is inherited from an interface but is not declared neither in this class nor in its base classes.

In both cases the method is an abstract method in the class QualT. For this case DecC cannot be computed, but we know for sure that ContextT for an abstract method is instance. Also, because the method was declared in an interface, it is certainly public.

RULE INVOKE-COMPUTE-METHODINFO-UNMAPPED-METHOD-CONTEXTTYPE

$$\left\langle \_\ \texttt{.}\ \texttt{methodInfo}\ (\mathit{Sig}, \mathit{QualC}, \frac{\cdot_K}{\mathit{MetaT}}, \_, \frac{\cdot_K}{\texttt{instance}}, \frac{\cdot_K}{\texttt{public}})(\_)\ \cdots \right\rangle_{\mathsf{k}}$$
$$\langle \mathit{QualC} \rangle_{\mathsf{classType}}\ \ \langle \mathit{MetaT} \rangle_{\mathsf{classMetaType}}\ \ \langle \mathit{Methods} \rangle_{\mathsf{methods}}$$

requires $\neg_{\mathit{Bool}} \mathit{Sig}\ \texttt{in}\ \texttt{keys}\ (\mathit{Methods})$

The last rule of step 3 loads ContextT and Acc. It requires DecC, so this rule may only match after the second rule for methodInfo().

RULE INVOKE-COMPUTE-METHODINFO-CONTEXTTYPE

$$\left\langle \_ \;.\; \texttt{methodInfo}\,(\mathit{Sig}, \_, \_, \mathit{DecC}, \frac{\cdot_K}{\mathit{ContextT}}, \frac{\cdot_K}{\mathit{Acc}})(\_)\ \cdots \right\rangle_{\mathsf{k}}$$
$$\langle \mathit{DecC} \rangle_{\mathsf{classType}} \quad \langle \mathit{Sig} \rangle_{\mathsf{methodSignature}} \quad \langle \mathit{ContextT} \rangle_{\mathsf{methodContextType}} \quad \langle \mathit{Acc} \rangle_{\mathsf{methodAccessMode}}$$

1.5 Lookup method declaration

In the third step of the method invocation algorithm, the actual method declaration is chosen. This step starts once all the fields of methodInfo() were filled in (where possible). The rules of this step rewrite methodInfo() into methodRef() - another auxiliary data structure. The production methodRef() is a reference to a method declaration. It contains two fields - Sig and DecC - the signature and the declaration class. The implementation class is the class that contains the actual method declaration to be invoked.

$$\text{SYNTAX} \quad \mathit{MethodName} ::= \texttt{methodRef}\,(\mathit{Signature}, \mathit{RefType})$$

Since we already know the signature, this phase amounts to computing DecC. This step contains different rules for the following cases:

- Static method (JLS §15.12.4.4 paragraph 2)
- Instance method with target being null (JLS §15.12.4.4 paragraph 3)
- Instance method with non-null target, private method (JLS §15.12.4.4 paragraph 4)
- Instance method with non-null target, access mode is protected or public. This also includes qualifying type being interface. (JLS §15.12.4.4 paragraph 6 and point 1)
- Instance method with non-null target, access mode is package (no dedicated mention in JLS §15.12.4.4)

The method below is for the first case. If the method is static, then the declaring type DecC is the qualifying type. The qualifier is discarded by rewriting it into $\cdot_K$.

RULE INVOKE-METHODINFO-STATIC

$$\frac{\_}{\cdot_K} \;.\; \frac{\texttt{methodInfo}\,(\mathit{Sig}, \_, \texttt{class}, \mathit{DecC}, \texttt{static}, \_)}{\texttt{methodRef}\,(\mathit{Sig}, \mathit{DecC})}(\_)$$

If the qualifier value is null and ContextT is instance, then NullPointerException is thrown and method invocation expression is discarded. It is only at this point that we should check the qualifier whether it is null or not. If ContextT is static, then the previous rule will match, and no exception will be thrown.

RULE INVOKE-METHODINFO-INSTANCE-ON-NULL

$$\frac{\texttt{null :: \_ . methodInfo}\,(\_, \_, \_, \_, \texttt{instance}, \_)(\_)}{\texttt{throw new NullPointerException(null :: String) ;}}$$

The logic for private instance methods is the same as for static methods, with the difference that the qualifier is not discarded.

RULE INVOKE-METHODINFO-INSTANCE-PRIVATE

$$\texttt{objectRef}\,(\_, \_)\ \texttt{::}\ \_ \;.\; \frac{\texttt{methodInfo}\,(\mathit{Sig}, \_, \texttt{class}, \mathit{DecC}, \texttt{instance}, \texttt{private})}{\texttt{methodRef}\,(\mathit{Sig}, \mathit{DecC})}(\_)$$

If the method is protected or public, then we should call the version of the method visible to the runtime type of the qualifying object (ObjC). Recall that the runtime type of an object is stored in the second argument of objectRef(). This case also covers qualifying type interface, since interface methods are always public. The right method will always be the one referred by the signature Sig in the cell $\langle\rangle_{\mathsf{methods}}$ associated with the actual object class. This is because the unfolding phase populates $\langle\rangle_{\mathsf{methods}}$ with the union of methods inherited from the base class and methods declared in the current class, the latter overriding the former. The variable DecC is the class where the right method version is declared.

RULE INVOKE-METHODINFO-INSTANCE-PROTECTED-OR-PUBLIC

$$\left\langle \texttt{objectRef}\,(\_, ObjC) :: \_ \;.\; \frac{\texttt{methodInfo}\,(Sig, \_, \_, \_, \texttt{instance}, Acc)}{\texttt{methodRef}\,(Sig, DecC)}(\_) \cdots \right\rangle_{\mathsf{k}}$$

$\langle ObjC \rangle_{\mathsf{classType}}$ $\langle \cdots\ Sig \mapsto DecC\ \cdots \rangle_{\mathsf{methods}}$

requires $Acc =_K$ `protected` $\vee_{Bool}$ $Acc =_K$ `public`

The most complex case is for instance methods with package access mode. The precise semantics of overriding for all access modes is defined in JLS §8.4.6.1:

An instance method derivedM declared in a class Derived overrides another method with the same signature, baseM, declared in class Base iff both:

1. Derived is a subclass of Base.
2. Either
   a. baseM is public, protected, or declared with package access in the same package as derivedM
   b. derivedM overrides a method middleM, middleM distinct from baseM and derivedM, such that middleM overrides baseM

The transitive rule for overriding relation (2b) is required specifically for package access mode. Consider the following example:

```java
package a;
public class A {
  void f(int a) { ... }
}

package a;
public class B extends A {
  protected void f(int a) { ... }
}

package b;
import a.*;
public class C extends B {
  protected void f(int a) { ... }
}

((A) new C()).f();
```

The method in class C overrides the method in class A transitively through the method in B. There is no direct overriding between A and C, because the method is declared with default (package) access mode in A, and class C is in a different package. Note that if the access mode in B would have been package instead of protected, there would be no overriding.

In order to correctly handle such cases we have to analyse all the classes in the inheritance chain between the qualifying type and the qualifier runtime type.

The algorithm employed in K-Java is significantly different from the one in JLS, but it is much simpler to implement. Yet it yields the correct behaviour and was extensively tested by our test suite. The JLS algorithm involves starting the search from the runtime type of the qualifier and moving upwards in the inheritance tree until we find the FIRST method that overrides the originally called method (or is the originally called method itself). This apparently simple algorithm leads to multiple particular cases when we consider the transitive rule (2b above) for overriding.

In contrast, the K-Java algorithm starts the search with the qualifying type (e.g. static type of the qualifier expression) and moves downwards in the inheritance chain until it reaches the runtime type of the qualifier. When all classes in the chain were traversed the algorithm returns the LAST found method (e.g. defined in the most derived class) that overrides the original one.

The rule for package access mode delegates searching for the right method declaration to the auxiliary function lookupPM(). The function takes 3 arguments:

- method signature Sig
- the list of classes in the inheritance chain between the qualifying class QualC and the actual object class ObjC. This list is produced by classChain()
- the third argument represents the declaring class of the best method found so far. It is initialized with $\cdot_K$.

RULE INVOKE-METHODINFO-INSTANCE-PACKAGE

$$\left\langle \texttt{objectRef}\,(\_, ObjC) :: QualC \;.\; \frac{\texttt{methodInfo}\,(Sig, QualC, \texttt{class}, \_, \texttt{instance}, \texttt{package})}{\texttt{lookupPM}\,(Sig,\ \texttt{classChain}\,(QualC, ObjC), \cdot_K)}(\_) \cdots \right\rangle_{\mathsf{k}}$$

$$\langle QualC \rangle_{\mathsf{classType}} \quad \langle \cdots\ Sig \mapsto \_\ \cdots \rangle_{\mathsf{methods}}$$

Before the evaluation of lookupPM() may begin, the term lookupPM() has to be heated to the top of computation. The side condition in the context rule below ensured that the second argument of method call expression is heated only if it contains a term lookupPM(). If it has other forms, such as the method name or methodInfo(), it won't be heated.

CONTEXT
$\_ \;.\; \square(\_)$
requires $\texttt{getKLabel}\,(\square) =_{KLabel} {}'lookupPM$

SYNTAX $K ::= \texttt{classChain}\,(ClassType, ClassTypes)$

SYNTAX $K ::= \texttt{lookupPM}\,(Signature, ClassTypes, K)$ [strict(2,3)]

The rules for lookupPM() are based on the following two properties of the configuration:

- if the cell $\langle\rangle_{\mathsf{methods}}$ for a particular class contains a key Sig, then $\langle\rangle_{\mathsf{methods}}$ for all classes derived from it will contain the key Sig.
- if a particular class contain a method declaration with signature Sig access mode Acc, then all declarations of Sig in derived classes (that are not necessarily overriding!) will have the access mode equal to either Acc or a value wider than Acc.

The first property is ensured by the unfolding algorithm. Because $\langle\rangle_{\mathsf{methods}}$ of a derived class inherit all the $\langle\rangle_{\mathsf{methods}}$ of the direct base class, the map $\langle\rangle_{\mathsf{methods}}$ may only grow from base classes to derived. The second property is ensured by restrictions on overriding specified in JLS §8.4.8.3: "The access modifier (§6.6) of an overriding or hiding method must provide at least as much access as the overridden or hidden method".

The search for the right package method declaration is performed from the base-most class in the chain (the left-most one) to the most derived one. Every rule matches and deletes the leftmost class in the class chain (CurrentC), and possibly rewrites the third argument into the current class. The first rule matches when there is no declaring class yet (third argument is $\cdot_K$, the initial case).

RULE LOOKUPPM-LAYER-FIRST-DEC-FOUND

$$\left\langle \texttt{lookupPM}\,(\mathit{Sig}, \frac{\mathit{CurrentC}\;\texttt{,}\;\mathit{Cs}}{\mathit{Cs}}, \frac{\cdot_K}{\mathit{DecC}})\;\cdots \right\rangle_{\mathsf{k}}$$
$$\langle \mathit{CurrentC} \rangle_{\mathsf{classType}} \;\; \langle \cdots\; \mathit{Sig} \mapsto \mathit{DecC}\; \cdots \rangle_{\mathsf{methods}}$$

The second rule matches when we already found a declaring class (OldDecC) and the current class CurrentC has another method declaration with the right signature. The presence of a declaration with signature Sig inside CurrentC is identified by the match $\langle\langle \mathit{CurrentC}\rangle_{\mathsf{classType}}\;\langle \ldots \mathit{Sig} \mapsto \mathit{CurrentC} \ldots\rangle_{\mathsf{methods}}\rangle_{\mathsf{class}}$, according to the definition of $\langle\rangle_{\mathsf{methods}}$.

If the method in CurrentC directly overrides the method in OldDecC, the declaring class is updated to CurrentC. Otherwise the declaring class stays unchanged. The rules for direct overriding (case 1a above) are defined in the auxiliary function isOverridden(). The function takes three arguments:

- The base class OldDecC
- The derived class CurrentC
- The access mode Acc of the definition of Sig in OldDecC.

RULE LOOKUPPM-NEW-METHOD

$$\left\langle \texttt{lookupPM}\,(\mathit{Sig}, \frac{\mathit{CurrentC}\;\texttt{,}\;\mathit{Cs}}{\mathit{Cs}}, \frac{\mathit{OldDecC}}{\texttt{if (isOverridden}\,(\mathit{OldDecC}, \mathit{Acc}, \mathit{CurrentC}))\;\{\mathit{CurrentC}\}\;\texttt{else}\;\{\mathit{OldDecC}\}})\;\cdots \right\rangle_{\mathsf{k}}$$
$$\langle\langle \mathit{OldDecC}\rangle_{\mathsf{classType}}\;\langle \mathit{Sig}\rangle_{\mathsf{methodSignature}}\;\langle \mathit{Acc}\rangle_{\mathsf{methodAccessMode}}\;\cdots\rangle_{\mathsf{class}}$$
$$\langle\langle \mathit{CurrentC}\rangle_{\mathsf{classType}}\;\langle \cdots\; \mathit{Sig} \mapsto \mathit{CurrentC}\;\cdots\rangle_{\mathsf{methods}}\;\cdots\rangle_{\mathsf{class}}$$

SYNTAX $K ::= \texttt{isOverridden}\,(\mathit{ClassType}, \mathit{AccessMode}, \mathit{ClassType})$

RULE
$$\frac{\texttt{isOverridden}\,(\_, \texttt{public}, \_)}{\mathsf{true}}$$
RULE
$$\frac{\texttt{isOverridden}\,(\_, \texttt{protected}, \_)}{\mathsf{true}}$$
RULE
$$\frac{\texttt{isOverridden}\,(\mathit{BaseC}, \texttt{package}, \mathit{SubC})}{\texttt{getPackage (getTopLevel}\,(\mathit{BaseC}))\;\texttt{==}\;\texttt{getPackage (getTopLevel}\,(\mathit{SubC}))}$$
RULE
$$\frac{\texttt{isOverridden}\,(\_, \texttt{private}, \_)}{\mathsf{false}}$$

The third rule represents the case when CurrentC chain does not contain method declarations with signature Sig. This case is identified by the side condition CurrentC =/=K DecC. Indeed, the two classes are different only when the entry Sig $\mapsto$ DecC in $\langle\rangle_{\mathsf{methods}}$ was inherited rather than produced by a method in CurrentC.

RULE LOOKUPPM-NO-NEW-METHOD

$$\left\langle \texttt{lookupPM}\,(Sig, \underset{Cs}{\underline{CurrentC\,,\,Cs}}, \_)\ \cdots \right\rangle_{\mathsf{k}}$$

$$\langle CurrentC \rangle_{\mathsf{classType}}\ \langle \cdots\ Sig \mapsto DecC\ \cdots \rangle_{\mathsf{methods}}$$

requires $CurrentC \neq_K DecC$

The last rule matches when the chain of classes stored in the first argument remains empty. It rewrites the whole lookupPM() into a reference to the method that has to be invoked.

RULE LOOKUPPM-END

$$\frac{\texttt{lookupPM}\,(Sig, \cdot_{ClassTypes}, DecC)}{\texttt{methodRef}\,(Sig, DecC)}$$

For the code example above, the term lookupPM() will pass through the following forms during evaluation:

$\langle lookupPM(f(), (a.A, a.B, b.C), \cdot_K)\ldots\rangle_{\mathsf{k}}$ $\langle A\rangle_{\mathsf{classType}}$ $\langle\ldots f() \mapsto A\ldots\rangle_{\mathsf{methods}}$	rule 1
$\langle lookupPM(f(), (a.B, b.C), a.A)\ldots\rangle_{\mathsf{k}}$ $\langle\ \langle a.A\rangle_{\mathsf{classType}}\ \langle f()\rangle_{\mathsf{methodSignature}}\ \langle package\rangle_{\mathsf{methodAccessMode}}\cdots\rangle_{\mathsf{class}}$ $\langle\ \langle a.B\rangle_{\mathsf{classType}}\ \langle\ldots f() \mapsto a.B\ldots\rangle_{\mathsf{methods}}\cdots\rangle_{\mathsf{class}}$	rule 2, if returns true
$\langle lookupPM(f(), (b.C), a.B)\ldots\rangle_{\mathsf{k}}$ $\langle\ \langle a.B\rangle_{\mathsf{classType}}\ \langle f()\rangle_{\mathsf{methodSignature}}\ \langle protected\rangle_{\mathsf{methodAccessMode}}\cdots\rangle_{\mathsf{class}}$ $\langle\ \langle b.C\rangle_{\mathsf{classType}}\ \langle\ldots f() \mapsto b.C\ldots\rangle_{\mathsf{methods}}\cdots\rangle_{\mathsf{class}}$	rule 2, if returns true
$\langle lookupPM(f(), \cdot_{ClassTypes}, b.C)\ldots\rangle_{\mathsf{k}}$	rule 3
$\langle methodRef(f(), b.C)\ldots\rangle_{\mathsf{k}}$	

When the term lookupPM() is first produced it takes as arguments the method signature (rendered here as f() for convenience), the chain of classes from the qualifying class A to the runtime class C, and $\cdot_K$ as the third argument. Since the third argument is $\cdot_K$ only the third rule can match. This rule deletes A from the class chain and updates the third argument to the class that defines the version of f() accessible to A. That class is A. For classes B and C the second rule for lookupMethodM() matches. In both cases the method f() defined in B and C overrides the previously found one. In the first case classes the method B.f() overrides A.f() because the access mode is package and both A and B are in the same package. In the second case C.f() overrides B.f() because B.f() have protected access mode, and is thus always overridden. The final result of method lookup procedure is the version of method f() declared in the class C.

1.6 Actual method invocation

The central rule of method invocation is matched when the second argument of method call expression reaches the form methodRef(). This rule performs the following operations:

- saves the rest of computation (RestK) and the content of $\langle\rangle_{\mathsf{methodContext}}$ as a new entry of the cell $\langle\rangle_{\mathsf{stack}}$ This data is restored back by the rules for return statement.
- Initializes the new method context.
  - The local variable environment $\langle\rangle_{\mathsf{env}}$ is emptied
  - current class $\langle\rangle_{\mathsf{crntClass}}$ is initialized to the class declaring the method
  - object location $\langle\rangle_{\mathsf{location}}$ is initialized to the location of the qualifier object for instance methods, or $\cdot_K$ for static methods. The extraction of the location from the qualifier value is performed by the function getOId().
- Rewrites the method call expression into a sequence of four terms:

– static initialization of the qualifying class
– parameters initialization
– actual method body
– a return statement with no arguments after the method body.

The function staticInit() triggers static initialization of the qualifying class, if this class was not initialized yet. Repeated calls of this function have no effect. Is required just for static methods and is described in JLS §12.4. For an instance method call, the qualifying class will always be initialized already, so staticInit() will have no effect.

The function initParams() rewrites each parameter declaration into two statements. First is a local variable declaration with that parameter name. The second is an assignment to that variable of the actual argument value.

The return statement at the end ensures that there is a return statement on every execution path of the method. The statement will only be useful for methods with return type void, as methods returning a value are required by JLS to have an appropriate return statement on every return path.

RULE INVOKE-METHODREF

$$\left\langle \frac{Qual\ .\ \texttt{methodRef}\ (Sig, DecC)(Args) \curvearrowright RestK}{\texttt{staticInit}\ (DecC) \curvearrowright \texttt{initParams}\ (Params, Args) \curvearrowright Body \curvearrowright \texttt{return}\ ;} \right\rangle_{\mathsf{k}}$$

$$\left\langle \frac{\cdot_{List}}{(RestK, MethodContext)} \cdots \right\rangle_{\mathsf{stack}} \left\langle \frac{MethodContext}{\langle \cdot_{Map} \rangle_{\mathsf{env}}\ \langle DecC \rangle_{\mathsf{crntClass}}\ \langle \texttt{getOId}\ (Qual) \rangle_{\mathsf{location}}} \right\rangle_{\mathsf{methodContext}}$$

$$\langle DecC \rangle_{\mathsf{classType}}\ \langle Sig \rangle_{\mathsf{methodSignature}}\ \langle Params \rangle_{\mathsf{methodParams}}\ \langle Body \rangle_{\mathsf{methodBody}}$$

SYNTAX $K ::= \texttt{getOId}\ (K)$ [function]

RULE

$$\frac{\texttt{getOId}\ (\ \texttt{objectRef}\ (OId, \_)\ ::\ \_)}{OId}$$

RULE

$$\frac{\texttt{getOId}\ (\cdot_K)}{\cdot_K}$$

SYNTAX $K ::= \texttt{initParams}\ (Params, TypedVals)$

RULE INITPARAMS

$$\frac{\texttt{initParams}\ (\{T\ \ X\}, RestP, (TV\ , RestV))}{T\ \ X\ ; \curvearrowright (X = ((T)\,TV))\ ; \curvearrowright \texttt{initParams}\ (RestP, RestV)}$$

RULE INITPARAMS-END

$$\frac{\texttt{initParams}\ (\cdot_{Params}, \cdot_{TypedVals})}{\cdot_K}$$

[structural]

1.7 Conclusion

While maintaining a close correspondence between JLS and K-Java would be an interesting quest on its own, there are a number of reasons why such a goal would not be practical. Some of the reasons are:

- JLS specification describes not only the execution of correct Java programs, but also runtime checks that have to be performed to ensure the consistency of the bytecode, and errors that have to be thrown once this consistency is violated. Such bytecode inconsistencies may arise when someone compiles a program with one version of a library and tries to execute it with another version. Since in K-Java we perform the logic corresponding to compilation and execution at the same time, we cannot encounter such inconsistencies. This is why sections like §15.12.4.3, does not have a correspondent in K-Java.

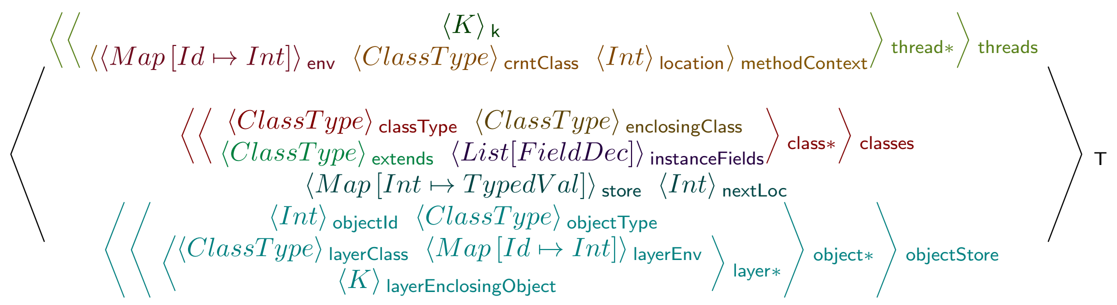
- Because K-Java operates directly over the source code of Java, with no other preprocessing than the unfolding phase, it carries less static information than the bytecode. For this reason some static information needs to be computed in K-Java each time a method is invoked. This is why semantics step 3 is needed. As we will see below, rules for this step are straightforward and consist of loading the right data from the right cells into an auxiliary data structure.
- Although JLS avoids references to bytecode as much as possible, sometimes it contains references to features specific to bytecode. For example a method call in JLS has an invocation mode associated with it, that might be static, nonvirtual, virtual, interface or others. Since this invocation mode is computed at compile-time, JLS runtime semantics is described separately for each such invocation mode. In K, since we don't have such a classification by invocation modes, we often have fewer cases.
- The logic of exception propagation is repeated in JLS in every context an where an exception might interrupt the usual execution flow. At the same time K abstractions allow us to cover all those cases by a fixed set of rules.
- Most complex parts of JLS prose have the form if ... otherwise if .. otherwise. While matching the condition under an if can be done by a single rule in K, matching the negation of a condition is more complex and may involve many rules. This difficulty arises from the fact that K does not offer built-in support to express lack of a match of a particular rule as a side condition of another rule.
- The powerful mechanism of strictness in K allows us to seamlessly define many language features that have many corresponding lines of text of JLS. This is especially true for order of evaluation and exception propagation.

On overall, we believe that the lack of direct correspondence between JLS and K rules is not a disadvantage. By relaxing this correspondence we were able to produce a semantics that is more concise than JLS. While K-Java cannot achieve the same level of ease of reading as JLS, it might serve as a complementary reference. A formal semantics definition might be useful to clarify the most ambiguous and technically complex parts of the semantics, such as rules of package method overriding that were presented above.

2 Object Instantiation

2.1 Background

In this subsection we present the fragment of configuration used by runtime method invocation. The figure below contains the cells and their sorts.



$\langle K\rangle_{k}$
$\langle\langle Map[Id \mapsto Int]\rangle_{env}$ $\langle ClassType\rangle_{crntClass}$ $\langle Int\rangle_{location}\rangle_{methodContext}$
thread*
threads
$\langle ClassType\rangle_{classType}$ $\langle ClassType\rangle_{enclosingClass}$
$\langle ClassType\rangle_{extends}$ $\langle List[FieldDec]\rangle_{instanceFields}$
class*
classes
$\langle Map[Int \mapsto TypedVal]\rangle_{store}$ $\langle Int\rangle_{nextLoc}$
$\langle Int\rangle_{objectId}$ $\langle ClassType\rangle_{objectType}$
$\langle ClassType\rangle_{layerClass}$ $\langle Map[Id \mapsto Int]\rangle_{layerEnv}$
$\langle K\rangle_{layerEnclosingObject}$
layer*
object*
objectStore
T

The cell $\langle\rangle_{k}$ stores the current computation. Inside $\langle\rangle_{env}$ we store the local environment – a map from variable names to their locations in the store. The cell $\langle\rangle_{methodContext}$ store information about the current object – the one accessible through the keyword this. Both $\langle\rangle_{\mathsf{env}}$ and $\langle\rangle_{\mathsf{methodContext}}$ play a special role in object instantiation.

The cell $\langle\rangle_{\mathsf{class}}$ contains various sub-cells holding the content of that class. The first cell in $\langle\rangle_{\mathsf{classType}}$ of sort ClassType that holds the fully qualified class name. This cell is a unique identifier of a class, and is used as a key to access other cells inside a $\langle\rangle_{\mathsf{class}}$. Next relevant cells inside $\langle\rangle_{\mathsf{class}}$ are $\langle\rangle_{\mathsf{enclosingClass}}$ - the directly enclosing class in case this class is an inner class. The vase class is stored inside $\langle\rangle_{\mathsf{extends}}$ and the list of declarations of instance fields without identifiers is stored in $\langle\rangle_{\mathsf{instanceFields}}$.

The next two cells are related to the store. The cell $\langle\rangle_{\mathsf{store}}$ have a central role in the semantics – it is the map from object locations (values in the cell $\langle\rangle_{\mathsf{env}}$) to their actual typed values. The cell $\langle\rangle_{\mathsf{nextLoc}}$ is the counter of store locations.

The remaining big group of cells – $\langle\rangle_{\mathsf{objectStore}}$ contains the inner structure of objects. The $\langle\rangle_{\mathsf{objectId}}$ is an unique identifier of the object. Every reference to this object in the store is a reference to this id. Inside $\langle\rangle_{\mathsf{objectType}}$ is the actual runtime type of the object. Next we have a list of $\langle\rangle_{\mathsf{layer}}$ cells, each of them representing an inheritance layer of the object. Starting from class Object and ending with the actual object type. Inside each layer $\langle\rangle_{\mathsf{layerClass}}$ stores its associated class, $\langle\rangle_{\mathsf{layerEnv}}$ – the fields and $\langle\rangle_{\mathsf{layerEnclosingObject}}$ – the enclosing object, in the case when $\langle\rangle_{\mathsf{layerClass}}$ is a non-static inner class. The complex rules for Java inner classes allow each layer to have its distinctive enclosing object, and we have tests that specifically target this requirement.

2.2 New instance creation

When a new instance creation expression reaches the top of computation, first it is normalized to a standard form. If it is an unqualified expression, an empty qualifier is added. Second, if the class to be instantiated is a simple name, it have to be converted to a fully qualified class name. At this stage this could only happen for true inner classes, and the fully qualified name is computed by concatenating the type of the qualifier and the class simple name, by the rule below.

RULE QUALIFIED-NEW-INSTANCE-RESOLVE-CLASS

$$Qual\ .\ \mathtt{new}\ \frac{Name}{\mathtt{getClassType}\ (\ \mathtt{toPackage}\ (\ \mathtt{typeOf}\ (Qual)),\ Name)}(\_)$$

After the new instance expression have been normalized, the qualifier and the arguments are brought to the top of computation by the strictness rules and evaluated. Qualifier is evaluated first, and arguments are evaluated left-to-right according to JLS.

When all the subexpressions of new have been evaluated, the main rule for new could apply. This rule touches a large number of cells, that will be explained next. First the current value of the counter inside $\langle\rangle_{\mathsf{nextLoc}}$ is used as the location of the newly created object. The counter is incremented for the next use. Inside $\langle\rangle_{\mathsf{objectStore}}$ a new cell $\langle\rangle_{\mathsf{object}}$ is created for the new object. For now it have just two sub-cells specified – $\langle\rangle_{\mathsf{objectId}}$ and $\langle\rangle_{\mathsf{objectType}}$, and no layers. Curiously we don't have to specify neither $\langle\rangle_{\mathsf{object}}$ nor $\langle\rangle_{\mathsf{objectStore}}$ cells explicitly here, we have to specify just the cells inside them that are modified. The capability to ignore surrounding cells when they can be automatically inferred is called configuration abstraction, another K feature[**?**]. In the cell $\langle\rangle_{\mathsf{store}}$ a new entry is created with key being L and value - a reference to the newly created object in $\langle\rangle_{\mathsf{object}}$. The content of $\langle\rangle_{\mathsf{methodContext}}$ is reset to a default state. This default state is required by rules that are applied next.

Inside $\langle\rangle_{\mathsf{k}}$ the new instance expression is rewritten into a sequence of computations that will be executed by the following rules. The auxiliary function staticInit() triggers static initialization of the instantiated class, in case it was not triggered earlier. Next, the function create() populates the layers of the object inside $\langle\rangle_{\mathsf{object}}$ This also includes allocation of all instance fields, and their initialization to the default value. Field initializers are not executed yet. The function setEncloser() sets the enclosing object for the current class, if the current class is an inner class. If some of the base classes are also inner classes, the encloser for their respective $\langle\rangle_{\mathsf{layer}}$ will be set as part of constructor invocation.

The next term in the computation (the one starting with typedLookup(L)) might look a bit weird, but it is in fact the invocation of the constructor. This term represents a mix of Java syntax for method invocation and auxiliary functions defined inside K-Java. It illustrates, among others, the power of K parser. Now, after all memory allocation procedures have been completed, it is the right time for it to be invoked. Preprocessing semantics transforms all constructors into plain methods. The function typedLookup(L) is evaluated into the object stored at the location L, that will serve as a qualifier for constructor invocation. The function getConsName() converts the class name into the name of the constructor method. What remains is plain Java syntax for method invocation.

The last two terms bring computation to the state required to continue execution. Function restoreMethoContext() restores $\langle\rangle_{\mathsf{methodContext}}$ to the the state before object creation. The last term is the result value of the object instantiation expression.

RULE QUALIFIED-NEW-INSTANCE

$$\left\langle \frac{Qual\ \texttt{.}\ \texttt{new}\ Class(Args)}{\begin{array}{c}\texttt{staticInit}\ (Class) \curvearrowright \texttt{create}\ (Class)\\ \curvearrowright \texttt{setEncloser}\ (\ \texttt{typedLookup}\ (L), Class, Qual)\\ \curvearrowright \texttt{typedLookup}\ (L)\ \texttt{.}\ \texttt{getConsName}\ (Class)(Args)\ \texttt{;}\\ \curvearrowright \texttt{restoreMethContext}\ (MethContext) \curvearrowright \texttt{typedLookup}\ (L)\end{array}} \cdots \right\rangle_{\mathsf{k}}$$

$$\left\langle \cdots \frac{\cdot_{Map}}{L \mapsto \texttt{objectRef}\ (L, Class)\ \texttt{::}\ Class} \cdots \right\rangle_{\mathsf{store}} \quad \left\langle \frac{L}{L +_{Int} 1} \right\rangle_{\mathsf{nextLoc}} \quad \frac{\cdot_{Bag}}{\langle \langle L\rangle_{\mathsf{objectId}}\ \langle Class\rangle_{\mathsf{objectType}} \rangle_{\mathsf{object}}}$$

$$\left\langle \frac{MethContext}{\langle \cdot_{Map}\rangle_{\mathsf{env}}\ \langle \cdot_{K}\rangle_{\mathsf{crntClass}}\ \langle L\rangle_{\mathsf{location}}} \right\rangle_{\mathsf{methodContext}}$$

SYNTAX $K ::= \texttt{create}\ (ClassType)$

RULE CREATE

$$\left\langle \frac{\texttt{create}\ (Class)}{\texttt{create}\ (BaseClass) \curvearrowright \texttt{setCrntClass}\ (Class) \curvearrowright FieldDecs \curvearrowright \texttt{addEnvLayer}} \cdots \right\rangle_{\mathsf{k}} \quad \langle Class\rangle_{\mathsf{classType}}$$

$$\langle BaseClass\rangle_{\mathsf{extends}}\ \ \langle FieldDecs\rangle_{\mathsf{instanceFields}}$$

[structural]

RULE CREATE-EMPTY-DISCARD

$$\frac{\texttt{create}\ (\cdot_{K})}{\cdot_{K}}$$

[structural]

SYNTAX $K ::= \texttt{setCrntClass}\ (ClassType)$

RULE SETCRNTCLASS

$$\left\langle \frac{\texttt{setCrntClass}\ (Class)}{\cdot_{K}} \cdots \right\rangle_{\mathsf{k}} \quad \left\langle \frac{\_}{Class} \right\rangle_{\mathsf{crntClass}}$$

[structural]

SYNTAX $K ::= \texttt{addEnvLayer}$

RULE ADDENVLAYER

$$\left\langle \frac{\texttt{addEnvLayer}}{\cdot_{K}} \;\cdots \right\rangle_{\mathsf{k}} \quad \left\langle \frac{Env}{\cdot_{Map}} \right\rangle_{\mathsf{env}} \quad \langle Class \rangle_{\mathsf{crntClass}} \quad \langle OId \rangle_{\mathsf{location}}$$

$$\left\langle \langle OId \rangle_{\mathsf{objectId}} \; \frac{\cdot_{Bag}}{\langle \langle Class \rangle_{\mathsf{layerClass}} \; \langle Env \rangle_{\mathsf{layerEnv}} \; \cdots \rangle_{\mathsf{layer}}} \;\cdots \right\rangle_{\mathsf{object}}$$

[structural]

Sets the enclosing object for a given object.

SYNTAX $K ::= \texttt{setEncloser}\,(K, ClassType, K)$ [strict(1,3)]

RULE SETENCLOSER-VALUE

$$\left\langle \frac{\texttt{setEncloser}\,(\,\texttt{objectRef}\,(OId, \_)\; \texttt{::}\; \_, Class, EncloserVal\; \texttt{::}\; \_)}{\cdot_{K}} \;\cdots \right\rangle_{\mathsf{k}} \quad \langle OId \rangle_{\mathsf{objectId}} \quad \langle Class \rangle_{\mathsf{layerClass}}$$

$$\left\langle \frac{\_}{EncloserVal\; \texttt{::}\; EncloserClass} \right\rangle_{\mathsf{layerEnclosingObject}} \quad \langle Class \rangle_{\mathsf{classType}} \quad \langle EncloserClass \rangle_{\mathsf{enclosingClass}}$$

RULE SETENCLOSER-NOVALUE

$$\frac{\texttt{setEncloser}\,(\_, \_, \cdot_{K})}{\cdot_{K}}$$