A COMPLETE SEMANTICS FOR JAVA

BY

DENIS BOGDĂNAȘ

DISSERTATION
Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Alexandru Ioan Cuza University of Iași, 2015
Iași, Romania

Doctoral Committee:

Professor Dr. Dorel Lucanu, Supervisor, UAIC
Abstract

This thesis presents K-Java, the first complete formal semantics of Java 1.4, defined in K Framework. The semantics yields an interpreter and a model-checker for multithreaded programs. To test the completeness of K-Java, we developed our own suite of more than 840 tests that exercise every Java 1.4 feature, corner case or feature interaction. The tests were developed alongside K-Java, following Test Driven Development. In order to maintain clarity while handling the great size of Java, the semantics was split into two separate definitions – a static semantics and a dynamic semantics. The output of the static semantics is a preprocessed Java program, which is passed as input to the dynamic semantics for execution. The preprocessed program is a valid Java program, which uses a subset of the features of Java. The test suite and the static semantics may be regarded as side contributions, they are generic and ready to be used in other Java-related projects.
I would like to thank my adviser Dorel Lucanu for giving me a good research topic that suited me well, and for constantly pushing me to work harder. I believe his advising style helped me strengthen my self-discipline, and I shall consider this a side benefit from doing a PhD. I am also grateful to professor Grigore Roșu from UIUC, with whom I had the opportunity to collaborate during the hardest part of my work. I never would have made it this far without his moral support and advices.

Although we didn’t have the opportunity to meet, Chucky Ellison, the author of C semantics in $\mathbb{K}$, had a great influence on my work. His thesis was for me a model to follow, his success was a motivation to do the same. An experience having enabling role on my thesis was Programming Languages Mentoring Workshop (PLMW), collocated with POPL, which I attended twice. I thank PLMW organizers for this opportunity. Only when I saw the mountain with my own eyes I realized it is not that high, and climbing it was within my abilities.

I am thankful to my colleagues Andrei Arusoaie and Radu Mereuță, for answering my technical questions, for sharing their know-how with me, but also for giving me a sense of competition. I believe this also had a role in aiming a bit higher. I’m also thankful to Traian Şerbănuţă, for kind words when I needed them the most.

My friends Marius, Paulici, Vlad, Eugen, George and Razvan — thank you for sharing your company with me, for giving a sense of joy. In particular I’m grateful to Eugen Cazacu for our amazing trips to mountains that helped me recharge the batteries. I also thank my parents for being there for me.

Finally, the person who influenced me wanting to do special things in computer science was my high school teacher Ludmila Țurcanu. Thank you for being both strict an loving, and for introducing me to the world of computer science olympiads.
My research has been partially supported by Romanian DAK grant SMIS-CSNR 602-12516 and by US NSF grants CCF-1218605 and CCF-1318191.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Author’s Publications</td>
<td>1</td>
</tr>
<tr>
<td>Scholarships</td>
<td>2</td>
</tr>
<tr>
<td>List of Figures</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>1.1 Problem Context and Contribution</td>
<td>5</td>
</tr>
<tr>
<td>1.2 The Java Language Specification</td>
<td>8</td>
</tr>
<tr>
<td>1.3 A Flavor of Java: Static Typing and Access Modes</td>
<td>10</td>
</tr>
<tr>
<td>Chapter 2 Related Work</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Other Executable Semantics of Java</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Other Large-Scale Executable Semantics</td>
<td>16</td>
</tr>
<tr>
<td>Chapter 3 Introduction to K Framework</td>
<td>18</td>
</tr>
<tr>
<td>3.1 A K Tutorial</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1 IMP Language Syntax</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2 IMP Language Semantics</td>
<td>20</td>
</tr>
<tr>
<td>3.2 AST-Based Terms in K</td>
<td>28</td>
</tr>
<tr>
<td>Chapter 4 Parsing Java Programs</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 5 Static Semantics</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Transformations</td>
<td>34</td>
</tr>
<tr>
<td>5.2 Phases</td>
<td>38</td>
</tr>
<tr>
<td>Chapter 6 Dynamic Semantics</td>
<td>49</td>
</tr>
<tr>
<td>6.1 Configuration</td>
<td>49</td>
</tr>
<tr>
<td>6.2 Types and Values</td>
<td>51</td>
</tr>
<tr>
<td>6.3 Expressions</td>
<td>54</td>
</tr>
<tr>
<td>6.4 Statements</td>
<td>58</td>
</tr>
</tbody>
</table>
6.4.1 Statement if ........................................ 58
6.4.2 Statement Blocks .................................. 59
6.4.3 Statements while, break, continue .............. 59
6.4.4 Statements throw and try/catch ................. 62
6.5 Memory Model ......................................... 65
6.6 Variables ............................................. 67
   6.6.1 Local Variable Declaration ....................... 69
6.6.2 Local Variable Lookup ............................. 70
6.6.3 Field Lookup .................................... 70
6.6.4 Assignment Operator ............................... 72
6.7 New Instance Operator ............................... 75
6.8 Method Invocation ................................. 80
   6.8.1 Preliminaries .................................. 81
6.8.2 Evaluation of the Subexpressions ................. 82
6.8.3 Loading Method Information ..................... 82
6.8.4 Lookup Method Reference ......................... 84
6.8.5 Actual Method Invocation ......................... 93
6.9 Multithreading and Synchronization ................. 95
   6.9.1 Thread Startup and Termination ................. 96
6.9.2 Thread Synchronization ........................... 98
   6.9.3 Methods wait() and notify() .................. 100

Chapter 7 Applications ............................. 103
   7.1 State Space Exploration ......................... 103
   7.2 LTL Model-Checking ............................... 103

Chapter 8 Testing ................................. 108
   8.1 The Quest for a Test Suite ....................... 108
   8.2 Test Development Methodology .................... 109

Chapter 9 Conclusion ............................... 111
   9.1 Statistics ....................................... 111
   9.2 A Discussion on K ................................ 112
   9.3 Limitations ..................................... 114
   9.4 Future Work .................................... 114

Appendix A K-Java Dynamic semantics .............. 116
   A.1 Module CONFIGURATION-EXEC .................... 116
   A.2 Module EXPRESSIONS ............................. 120
      A.2.1 Boolean operators ............................ 120
      A.2.2 Numeric operators ........................... 122
      A.2.3 String operators ............................ 126
      A.2.4 Conditional operator ......................... 126
      A.2.5 Assignment operators ......................... 127
      A.2.6 Cast operator — primitive types ............. 128
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.10.3</td>
<td>Evaluation of the qualifier and the arguments</td>
<td>162</td>
</tr>
<tr>
<td>A.10.4</td>
<td>Loading method information</td>
<td>163</td>
</tr>
<tr>
<td>A.10.5</td>
<td>Lookup method declaration</td>
<td>164</td>
</tr>
<tr>
<td>A.10.6</td>
<td>Actual method invocation</td>
<td>170</td>
</tr>
<tr>
<td>A.10.7</td>
<td>Conclusion</td>
<td>172</td>
</tr>
<tr>
<td>A.11</td>
<td>Module METHOD-INVOKE-REST</td>
<td>173</td>
</tr>
<tr>
<td>A.11.1</td>
<td>Method information for arrays and strings</td>
<td>173</td>
</tr>
<tr>
<td>A.11.2</td>
<td>Superclass method access — <code>A.super(...)</code></td>
<td>173</td>
</tr>
<tr>
<td>A.11.3</td>
<td>Auxiliary functions</td>
<td>174</td>
</tr>
<tr>
<td>A.12</td>
<td>Module STATIC-INIT</td>
<td>174</td>
</tr>
<tr>
<td>A.13</td>
<td>Module API-CORE</td>
<td>176</td>
</tr>
<tr>
<td>A.13.1</td>
<td><code>System.in</code>, <code>System.out</code>, <code>Scanner</code></td>
<td>176</td>
</tr>
<tr>
<td>A.13.2</td>
<td><code>Class Object</code></td>
<td>178</td>
</tr>
<tr>
<td>A.13.3</td>
<td><code>Class String</code></td>
<td>179</td>
</tr>
<tr>
<td>A.13.4</td>
<td>Array clone</td>
<td>180</td>
</tr>
<tr>
<td>A.13.5</td>
<td>Class literal operator — <code>A.class</code></td>
<td>181</td>
</tr>
<tr>
<td>A.14</td>
<td>Module API-THREADS</td>
<td>181</td>
</tr>
<tr>
<td>A.14.1</td>
<td>Method <code>Thread.start()</code></td>
<td>182</td>
</tr>
<tr>
<td>A.14.2</td>
<td>Synchronized statement</td>
<td>182</td>
</tr>
<tr>
<td>A.14.3</td>
<td><code>Thread.join()</code></td>
<td>183</td>
</tr>
<tr>
<td>A.14.4</td>
<td>Methods <code>wait()</code> and <code>notify()</code> — core rules</td>
<td>183</td>
</tr>
<tr>
<td>A.14.5</td>
<td><code>Object.wait()</code> — additional</td>
<td>185</td>
</tr>
<tr>
<td>A.14.6</td>
<td><code>Object.notify()</code>, <code>Object.notifyAll()</code> — additional</td>
<td>185</td>
</tr>
<tr>
<td>A.14.7</td>
<td><code>Thread.interrupt()</code></td>
<td>187</td>
</tr>
<tr>
<td>A.14.8</td>
<td>Thread termination</td>
<td>187</td>
</tr>
<tr>
<td>A.14.9</td>
<td>Debug aids</td>
<td>187</td>
</tr>
<tr>
<td>A.15</td>
<td>Module UNFOLDING</td>
<td>188</td>
</tr>
<tr>
<td>A.16</td>
<td>Module TO-STRING</td>
<td>191</td>
</tr>
<tr>
<td>A.16.1</td>
<td>Debug helper functions</td>
<td>194</td>
</tr>
<tr>
<td>A.17</td>
<td>Module SYNTAX-CONVERSIONS</td>
<td>197</td>
</tr>
<tr>
<td>A.17.1</td>
<td>Method parameter</td>
<td>197</td>
</tr>
<tr>
<td>A.17.2</td>
<td>Method invocation</td>
<td>197</td>
</tr>
<tr>
<td>A.17.3</td>
<td>Local variable declaration</td>
<td>197</td>
</tr>
<tr>
<td>A.17.4</td>
<td>Cast</td>
<td>198</td>
</tr>
<tr>
<td>A.17.5</td>
<td>Identifier (name) expression</td>
<td>198</td>
</tr>
<tr>
<td>A.17.6</td>
<td>Syntactic lists</td>
<td>199</td>
</tr>
<tr>
<td>Appendix B</td>
<td>K-Java Common modules</td>
<td>202</td>
</tr>
<tr>
<td>B.1</td>
<td>Module CORE-SORTS</td>
<td>202</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Computation phases</td>
<td>202</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Values</td>
<td>203</td>
</tr>
<tr>
<td>B.1.3</td>
<td>Class and member attributes</td>
<td>204</td>
</tr>
<tr>
<td>B.1.4</td>
<td>Misc definitions</td>
<td>204</td>
</tr>
<tr>
<td>B.1.5</td>
<td>Random unsorted content, syntax converters</td>
<td>205</td>
</tr>
</tbody>
</table>

viii
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2</td>
<td>Module CORE-FUNCTIONS</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Core utility functions</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Utilities for general-purpose programming</td>
</tr>
<tr>
<td>B.2.3</td>
<td>ClassType functions</td>
</tr>
<tr>
<td>B.3</td>
<td>Module CORE-CLASSES</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Shortcuts for frequently used classes</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Auxiliary functions for packages</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Auxiliary functions for classes</td>
</tr>
<tr>
<td>B.4</td>
<td>Module PRIMITIVE-TYPES</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Integer value normalization</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Type normalization</td>
</tr>
<tr>
<td>B.5</td>
<td>Module SUBTYPING</td>
</tr>
<tr>
<td>B.5.1</td>
<td>Subtyping among primitive types</td>
</tr>
<tr>
<td>B.5.2</td>
<td>Subtyping among reference types</td>
</tr>
<tr>
<td>B.5.3</td>
<td>Subtyping lists of types</td>
</tr>
<tr>
<td>B.6</td>
<td>Module AUX-STRINGS</td>
</tr>
</tbody>
</table>

Appendix C  K-Java Static semantics

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>Module CONFIGURATION-PREP</td>
</tr>
<tr>
<td>C.2</td>
<td>Module PROCESS-TYPE-NAMES</td>
</tr>
<tr>
<td>C.2.1</td>
<td>Compilation units</td>
</tr>
<tr>
<td>C.2.2</td>
<td>Package declarations</td>
</tr>
<tr>
<td>C.2.3</td>
<td>Class declaration</td>
</tr>
<tr>
<td>C.2.4</td>
<td>Auxiliary constructs</td>
</tr>
<tr>
<td>C.3</td>
<td>Module PROCESS-COMP-UNITS</td>
</tr>
<tr>
<td>C.4</td>
<td>Module PROCESS-IMPORTS</td>
</tr>
<tr>
<td>C.5</td>
<td>Module PROCESS-CLASS-DECS</td>
</tr>
<tr>
<td>C.5.1</td>
<td>Initiate the resolving of class bases</td>
</tr>
<tr>
<td>C.5.2</td>
<td>Resolve bases</td>
</tr>
<tr>
<td>C.5.3</td>
<td>Processing after bases were resolved</td>
</tr>
<tr>
<td>C.5.4</td>
<td>Computing imports map</td>
</tr>
<tr>
<td>C.6</td>
<td>Module PROCESS-CLASS-MEMBERS</td>
</tr>
<tr>
<td>C.6.1</td>
<td>Triggering the processing of depending types</td>
</tr>
<tr>
<td>C.6.2</td>
<td>Initiating class processing</td>
</tr>
<tr>
<td>C.6.3</td>
<td>Inheriting base types</td>
</tr>
<tr>
<td>C.6.4</td>
<td>Method declarations</td>
</tr>
<tr>
<td>C.6.5</td>
<td>Constructor declarations</td>
</tr>
<tr>
<td>C.6.6</td>
<td>Instance fields and instance initializers</td>
</tr>
<tr>
<td>C.6.7</td>
<td>Static fields and static initializers</td>
</tr>
<tr>
<td>C.6.8</td>
<td>Compile-time constants</td>
</tr>
<tr>
<td>C.6.9</td>
<td>Other members</td>
</tr>
<tr>
<td>C.6.10</td>
<td>Functions for accessing member modifiers</td>
</tr>
<tr>
<td>C.7</td>
<td>Module CORE-PREPROCESSING</td>
</tr>
<tr>
<td>C.7.1</td>
<td>Class-related functions</td>
</tr>
</tbody>
</table>

ix
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.7.2</td>
<td>Method-related functions</td>
<td>259</td>
</tr>
<tr>
<td>C.8</td>
<td>Module ELABORATION-CORE</td>
<td>261</td>
</tr>
<tr>
<td>C.8.1</td>
<td>Elaboration phase — introduction</td>
<td>261</td>
</tr>
<tr>
<td>C.8.2</td>
<td>Core auxiliary definitions</td>
<td>262</td>
</tr>
<tr>
<td>C.8.3</td>
<td>Elaboration of code blocks</td>
<td>262</td>
</tr>
<tr>
<td>C.8.4</td>
<td>Auxiliary functions for elaboration phase</td>
<td>264</td>
</tr>
<tr>
<td>C.8.5</td>
<td>Elaboration of statements — the step elabEnd</td>
<td>265</td>
</tr>
<tr>
<td>C.8.6</td>
<td>Elaboration of KListWrap — the list of terms</td>
<td>266</td>
</tr>
<tr>
<td>C.8.7</td>
<td>Auxiliary functions for other modules</td>
<td>267</td>
</tr>
<tr>
<td>C.9</td>
<td>Module ELABORATION-TOP-BLOCKS</td>
<td>268</td>
</tr>
<tr>
<td>C.10</td>
<td>Module ELABORATION-STATEMENTS</td>
<td>271</td>
</tr>
<tr>
<td>C.10.1</td>
<td>Statements for, block and catch</td>
<td>271</td>
</tr>
<tr>
<td>C.10.2</td>
<td>Elaboration of catch parameters</td>
<td>272</td>
</tr>
<tr>
<td>C.10.3</td>
<td>Local variable declarations</td>
<td>272</td>
</tr>
<tr>
<td>C.10.4</td>
<td>Explicit constructor invocations — this(), super().</td>
<td>273</td>
</tr>
<tr>
<td>C.11</td>
<td>Module ELABORATION-TYPES</td>
<td>275</td>
</tr>
<tr>
<td>C.12</td>
<td>Module ELABORATION-EXPRESSIONS</td>
<td>277</td>
</tr>
<tr>
<td>C.12.1</td>
<td>Numeric, boolean and String operators</td>
<td>277</td>
</tr>
<tr>
<td>C.12.2</td>
<td>Conditional operator</td>
<td>279</td>
</tr>
<tr>
<td>C.12.3</td>
<td>Assignment operator</td>
<td>280</td>
</tr>
<tr>
<td>C.12.4</td>
<td>Cast operator</td>
<td>280</td>
</tr>
<tr>
<td>C.12.5</td>
<td>Expressions over reference types</td>
<td>282</td>
</tr>
<tr>
<td>C.13</td>
<td>Module ELABORATION-VARS</td>
<td>282</td>
</tr>
<tr>
<td>C.13.1</td>
<td>Unqualified variable references</td>
<td>282</td>
</tr>
<tr>
<td>C.13.2</td>
<td>Self-references: this and A.this</td>
<td>284</td>
</tr>
<tr>
<td>C.13.3</td>
<td>Fields</td>
<td>284</td>
</tr>
<tr>
<td>C.14</td>
<td>Module ELABORATION-METHOD-INVOKE</td>
<td>287</td>
</tr>
<tr>
<td>C.14.1</td>
<td>Regular method calls</td>
<td>287</td>
</tr>
<tr>
<td>C.14.2</td>
<td>Superclass method calls: super.m()</td>
<td>288</td>
</tr>
<tr>
<td>C.14.3</td>
<td>Method reference lookup</td>
<td>290</td>
</tr>
<tr>
<td>C.14.4</td>
<td>Method signature lookup</td>
<td>291</td>
</tr>
<tr>
<td>C.14.5</td>
<td>Method accessibility check</td>
<td>292</td>
</tr>
<tr>
<td>C.15</td>
<td>Module ELABORATION-NEW-INSTANCE</td>
<td>293</td>
</tr>
<tr>
<td>C.16</td>
<td>Module ELABORATION-ARRAYS</td>
<td>295</td>
</tr>
<tr>
<td>C.16.1</td>
<td>Desugaring of c-style array declarators</td>
<td>295</td>
</tr>
<tr>
<td>C.16.2</td>
<td>Main array-related expressions</td>
<td>297</td>
</tr>
<tr>
<td>C.17</td>
<td>Module ELABORATION-CATEGORIES</td>
<td>298</td>
</tr>
<tr>
<td>C.18</td>
<td>Module LITERALS</td>
<td>309</td>
</tr>
<tr>
<td>C.18.1</td>
<td>Auxiliary constructs</td>
<td>309</td>
</tr>
<tr>
<td>C.18.2</td>
<td>Integer literals</td>
<td>311</td>
</tr>
<tr>
<td>C.18.3</td>
<td>Float literals</td>
<td>312</td>
</tr>
<tr>
<td>C.18.4</td>
<td>Boolean literals</td>
<td>312</td>
</tr>
<tr>
<td>C.18.5</td>
<td>Char literals</td>
<td>312</td>
</tr>
</tbody>
</table>
Author’s Publications

Papers that represent the foundation of this thesis:


Scholarships


2. 2nd Programming Languages Mentoring Workshop (PLMW) collocated with POPL’13, 2013, Rome, Italy.

3. 3rd Programming Languages Mentoring Workshop (PLMW) collocated with POPL’14, 2014, San Diego, USA.

4. Internship at University of Illinois at Urbana-Champaign, Formal Systems Laboratory, April-July 2014, Urbana, USA.
List of Figures

1.1 Method overloading with access modes .................................. 11
1.2 Package access mode and method overriding .......................... 12

2.1 Completeness comparison of various semantics of Java .............. 17

3.1 An example IMP program, computing the sum from 1 to \( n \) ..... 21
3.2 The AST representation of the IMP program from Figure 3.1 ... 29

4.1 The A1Term representation of the IMP program from Figure 3.1 32

5.1 The high-level structure of K-Java ........................................ 35
5.2 Transformations performed by the static semantics ................. 36
5.3 The static semantics configuration ....................................... 39

6.1 The dynamic semantics configuration .................................... 50
6.2 Auxiliary syntax definitions for types and values ................... 52
6.3 Rules for selected expressions: \(+\), \(\text{cast}\), \(\_?:\_\), \instancesof\) 55
6.4 Rules and syntax for if statement ....................................... 58
6.5 Rules for block statement ............................................... 59
6.6 Rules for while, break and continue statements ..................... 60
6.7 Rules for try/catch and throw ......................................... 63
6.8 The fragment of the configuration representing the memory model and cells used for object instantiation .......... 66
6.9 An example fragment of configuration containing inheritance and an object instance .............................................. 68
6.10 Rule for local variable declaration ..................................... 69
6.11 Rules for local variable lookup ........................................... 70
6.12 Rules for field lookup .................................................... 71
6.13 Rules for assignment ..................................................... 73
6.14 Normalization of new instance operator and evaluation of the arguments ................................................................. 76
6.15 Main rules for new instance operator .................................. 78
6.16 Loading method information - syntax and rules ..................... 85
6.17 Overriding example with package access mode ....................... 87
6.18 Lookup method reference — syntax and rules ....................... 89
6.19 Lookup method reference for package access mode ................ 91
6.20 Syntax and rules for isAccessible() .......................... 92
6.21 The evaluation sequence of lookupPackageMethod() for
    method f() from Figure 6.17 .................................. 93
6.22 Rules for actual method invocation .......................... 94
6.23 Code and rules for thread creation and thread termination .. 97
6.24 Rules for synchronized statement ............................ 99
6.25 Key rules for Object.wait() and Object.notify() ......... 101

7.1 A two-threaded blocking queue ............................... 105
Chapter 1

Introduction

This thesis presents K-Java, the first complete formal semantics of Java 1.4, defined in K Framework. The semantics yields an interpreter and a model-checker for multithreaded programs. Lacking an adequate test suite, we developed our own, with more than 840 tests that exercise every Java 1.4 feature, corner case or feature interaction. This test suite serves as assessment for K-Java completeness.

In this chapter, in Section 1.1 we explain the problem context and enumerate our contributions. At the end of the section we also give an outline for the rest of the thesis. In Section 1.2 we present the Java Language Specification and enumerate many of Java features, all covered by our semantics. In Section 1.3 a few of these features are illustrated in detail on code examples.

1.1 Problem Context and Contribution

Java is the second most popular programming language (http://langpop.com/), after C and followed by PHP. Both C and PHP have recently been given formal semantics [18, 20]. Like the authors of the C and PHP semantics, and many others, we firmly believe that programming languages must have formal semantics. Moreover, the semantics should be public and easily accessible, so inconsistencies are more easily spotted and fixed, and formal analysis tools should be based on such semantics, to eliminate the semantic gaps and thus errors in such tools. Without a formal semantics it is impossible to state or prove anything about the language with certainty, including that a program meets its specification, that a type system is sound, or that a compiler or interpreter is correct. While all analysis tools or implementations for the language invariably incorporate some variant of the language semantics, or a projection of it, these are hard to access and thus to assess.

To the best of our knowledge, the most notable attempts to give Java a
formal semantics are ASM-Java [48], which uses abstract state machines, and JavaFAN [19], which uses term rewriting. However, as discussed in Chapter 2 these semantics are far from being complete or even well tested. Each comes with a few sample Java programs illustrating only the defined features, and each can execute only about half of the other’s programs.

We present K-Java [8], a semantics for Java which systematically defines every single feature listed in the official definition of Java 1.4, which is the Java Language Specification, 2nd edition (JLS) [27], a 456-page 18-chapter document. Moreover, our semantics is thoroughly tested. In fact, we spent about half the time dedicated to this project to write tests, which are small Java programs exercising special cases of features or combinations of them. Specifically, we followed a Test Driven Development methodology to first develop the tests for the feature to be defined and interactions of it with previous features, and then defined the actual semantics of that feature. This way we produced a comprehensive set of 840 tests, which serves as a conformance test suite not only for our semantics, but also for testing various other Java tools. Considering that no such conformance test suite exists for Java, our tests can also be regarded as a contribution made by this thesis.

As a semantic framework and development tool for our Java semantics we chose K [23, 43]. There are several appealing aspects of K that made it suitable for such a large project. K provides a convenient notation for modular semantics of languages, as well as automatically-generated execution and formal analysis tools for the defined languages, such as a parser and interpreter, state-space explorer for reachability, and model-checker. The C [18] and PHP [20] semantics mentioned above have both been defined in K, and demonstrated their usefulness using some of the generic K tools. More advanced tools are under active development: symbolic execution engine, deductive program verifier and a translator of semantics to Coq [36]. Support for deductive verification will be provided by matching and reachability logic [42, 46, 49], making it unnecessary to define multiple semantics for the same language, e.g. an operational semantics for execution and an axiomatic semantics for program verification. At the time K-Java was published [8], Hoare-style deductive verification was demonstrated just on a core subset of C [24]. Only after the work on K-Java was complete, the same demonstration was made on the full JavaScript semantics [39].

To emphasize that our Java semantics is useful beyond just providing a
reference model for the language, we show how the builtin model-checker of K can be used to model-check multi-threaded Java programs. While this illustrates only one possible application of the semantics, other applications have the potential to be similarly derived from the language-independent tools that are under development as part of K.

Besides such immediate applications, we believe that our executable semantics of Java is also a convenient platform for experimenting with Java extensions. For example, \[3\] proposes to extend Java with traits and records, \[21\] with mixins, and \[29\] with immutable objects. The proposal to extend Java with generic types \[10\] made it to the language in Java 5. The widely debated lambda expression feature, with at least three evolving proposals \[33\], was finally incorporated in Java 8. Such extensions are easy to add to our semantics, and thanks to K one would immediately benefit not only from a reference model for them but also from all the formal analysis tools automatically offered by K.

**Contributions** The specific contributions of this thesis are:

- K-Java, the first complete semantics of Java 1.4, including multi-threading. More generally, K-Java is the first complete semantics for an imperative statically typed object-oriented concurrent language. In order to maintain clarity while handling the semantics great size we split the semantics into two parts, pipelined together: static semantics (Chapter 5) and dynamic semantics (Chapter 6).

- A demonstrative application — LTL model-checking of multithreaded programs (Chapter 7).

- A comprehensive test suite covering all Java constructs (Chapter 8).

- Application of the test suite to evaluate the completeness of other executable semantics of Java (Chapter 2).

- A language-independent Abstract Syntax Tree transformer, used to connect K framework to an external parser (Chapter 4).

The whole K-Java is included in the appendixes: dynamic semantics (Appendix A), common modules for static and dynamic semantics (Appendix B), static semantics (Appendix C) and syntax (Appendix D).
1.2 The Java Language Specification

Java is a statically, strongly typed, object-oriented, multi-threaded language. It is completely defined, i.e., unlike other languages [18, 4], it has no undefined or implementation-dependent features. Except for threads, Java is completely deterministic. The official definition of Java 1.4 is the JLS [27]. JLS has 456 pages and 18 chapters; the part that defines the language has 377 pages. Java is distributed as part of the Java Development Kit (JDK), together with a several thousand class library. The class library, however, is not part of the JLS. Nevertheless, we defined semantics to all classes that are central to the language and mentioned in the JLS, such as Object, String, Thread and a dozen exception types.

Challenges. The K-Java project faced formidable challenges. The first challenge is the sheer size of Java. At the imperative level the language has 36 operators (JLS §15) and 18 statements (§16). Java values may be subject to a large number of conversions (§5). There are complex rules for resolving an identifier (§6). More precisely, an identifier in Java could mean one of the following: package, class, field, local variable or method. Method name could be decided based on the immediate syntactic context alone. However, disambiguating between the remaining categories is non-trivial and involves many special cases. One of the most complex features of Java is method overloading, that we illustrate in the next subsection. Classes have complex initialization rules that include static initialization (§8.3.2, 12.4) and instance initialization (§8.3.2, 12.5). The matter is further complicated by the ability to call on the first line of a constructor either a constructor of the same class through this() or a constructor of the base class through super(). Interfaces interact with a wide number of features as they may have methods, static fields, specific rules for initialization and method overwriting.

Java has a number of modularity features, such as packages, imports, and four categories of nested classes: static inner, non-static inner, local and anonymous. Since nested classes may access names from their enclosing classes, they bring a large number of special cases for name resolving. Packages are important to define access modes, and access modes have challenging interactions with the other Java features, as will be illustrated in Section 1.3.

The separation of the whole semantics into static and dynamic definition is a consequence of Java being statically typed. Dynamically typed languages
like PHP or JavaScript need just a dynamic semantics. JLS clearly defines what computations should happen before the execution (compile-time) and what should happen during the execution (runtime). In Section 1.3 we present an example that illustrates the difficulties produced by static typing. While it might seem natural to have the two semantics for Java, we did not follow this approach from the beginning. How static typing influenced the design decisions of K-Java is discussed in Chapter 5.

Another challenge was the careful testing and implementation of all corner cases for each new feature. The difficulty arises when the new feature interacts with already defined features. For example, packages were among the last features added. We had to make sure packages were properly resolved in all contexts — in variable declarations, extends/implements clauses, cast expressions, etc. When we later added inner classes, we had to make sure inner classes work equally well in all the contexts above. For each context we had to test that inner classes might be referred both by simple names and by fully qualified names, that might contain package names. Our testing methodology is presented in Chapter 8.

Despite these challenges, we made no compromises and completely defined every single feature of Java 1.4.

**JLS content unrelated to K-Java.** Java was designed to be compiled into a bytecode format that is executed on a hardware-independent Java Virtual Machine (JVM). Consequently, some details of JLS deal specifically with the bytecode representation and are irrelevant here. Such parts are §12.1-12.3, 12.6-12.8 (details of JVM execution) and §13 (bytecode binary compatibility). JLS also defines all the compile-time errors that might be produced by incorrect programs. We do not cover them, as the focus of K-Java is to model the behavior of valid Java programs only.

Also, we do not cover Dynamic Class Loading (DCL), §12.2. Instead, in K-Java all classes are loaded at the beginning of the execution. JLS mentions the possibility to load classes at runtime, from other formats than bytecode, but sends the reader to the JVM specification for details. No other details are given. For this reason, it is fair to consider DCL a JVM rather than a Java feature, and to regard the default class loader (the one loading classes from “.class” files from the startup classpath) purely as a performance optimization, with no implication to the semantics.
1.3 A Flavor of Java: Static Typing and Access Modes

Every Java expression has a statically assigned type. Static types have various functions during execution:

- Subexpression types influence the type of the parent expression. For example, \( 1 + 2 \) is not the same as \( 1 + (\text{long})2 \).

- Integers’ type gives their precise representation within the allowed range. When the range for a certain type is exceeded, overflow occurs: \( 1000 \) is different from \( \text{(byte)}1000 \).

- When an object has a member (field or static method) that hides an inherited member one can pick the right member by manipulating the type of the qualifier. Given \( b \) of type \( B \), then \( b.v \) and \( (\text{A})b.v \) could refer to different fields.

- Method overloading allows methods with the same name and number of arguments, but with different argument types. Then \( f(0) \) and \( f((\text{long})0) \) may invoke different methods.

For most expressions, the static type might be computed at the same time as the actual value of the expression. One might think that static types could be computed during execution at the same time as the value of an expression is computed. We actually did this in an older version of the semantics, and it worked great for a while. However, we had to rethink this approach when the time came to define the ternary conditional operator \( _?:_ \). An expression \( a ? b : c \) evaluates to \( b \) when \( a \) is true, and to \( c \) otherwise. When \( b \) and \( c \) have different static types, the conditional expression will have the join type of the two[1]. Thus, the static type of the operator depends on the type of the two arguments, yet only one of these arguments is evaluated at runtime. It would be incorrect to compute the value of both \( b \) and \( c \) just for the sake of having their static type (due to possible side effects). Therefore the computation of static types had to be separated from the actual execution.

[1]JLS §15.25 puts some restrictions on the types of arguments of \( _?:_ \), that makes join in most cases to be the widest type of arguments 2 and 3.
class A {
    private String f(int a) { return "int";}  
    String f(long a) { return "long";}  

    String test() { 
        return f((byte)0); }  //int 
}  
class B { 
    String test() {  
        return new A().f((byte)0); }  //long 
} 

Figure 1.1: Method overloading with access modes

One of the most complex features of Java is method overloading. First, the arguments of a method call may have types that are different from the parameter types. When the method is overloaded, the version with the most specific types that are compatible with the argument types is invoked. Moreover, the choice of available versions of the method is influenced by access modes.

In Figure 1.1 both versions of f() are compatible with the call f((byte)0), yet the set of accessible versions is different for the two call sites. The call from A.test() has access to both versions, and it chooses the more specific one. However, the call from B.test() cannot access the private f(); it calls the one with default access mode. The situation is even more complex when overloading is combined with inheritance and subtype polymorphism.

One might be tempted to think that access modes are only required for static semantics and could be discarded before execution. In Java this is not the case. In Figure 1.2 the two methods f() and g() are very similar, yet the declaring class of the actually invoked methods is different. Since A.f() is declared with package access mode, and class C is in a different package, the method C.f() does not directly override A.f(). However, it does override A.f() indirectly, through the declaration B.f(). The method A.f() is accessible to B since both A and B are in the same package. As B.f() is declared with protected access mode, the method is accessible to any subclass of B, including to C. Thus, a link is established that allows C.f() to override A.f(). The very similar method g() is declared with the default (package) access mode in B, that prevents the connection between B.g() and
package a;
public class A {
   void f() { ... }
   void g() { ... }
}

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

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

A c = new C(); c.f(); //C.f()
c.g(); //B.g()

Figure 1.2: Package access mode and method overriding
\texttt{c.g()}. Hence, in order to determine the right method to be invoked, it is not sufficient to pick the last one visible to the dynamic (runtime) type of the qualifier. A correct semantics has to analyze the entire chain of classes between the static type of the qualifier and the dynamic type. We show how these rules are defined in K-Java in Section 6.8.

The examples above illustrate how static typing allows for a rich set of features, but it also brings significant complexity. Moreover, it is not enough to consider each language feature in isolation. It is required to define and test each combination that is potentially crosscutting.
Chapter 2

Related Work

Here we discuss two other major formal executable semantics of Java and compare them with K-Java. We also recall other large language semantics that influenced the design of K-Java.

2.1 Other Executable Semantics of Java

ASM-Java \[48\] is the first attempt to define a complete semantics of Java 1.0, and the most complete prior to K-Java, showing that it is feasible to define formal semantics to real-life programming languages. It was defined using abstract state machines (ASMs) and covers almost all major features of Java, except packages. ASM-Java is also executable; with the kind help of its authors we were able to make it work. ASM-Java comes with 58 tests that touch all their implemented features, which we used as one of our external test suites for K-Java. ASM-Java contains not only the semantics of Java, but also defines the compiler and the bytecode format, and gives a manual proof for the correctness of their compiler.

While K-Java uses a different formalism, it follows the ASM-Java methodology to separate the static and the dynamic semantics. Except for that, ASM-Java and K-Java are quite different. ASM-Java uses many auxiliary constructs in their preprocessed programs produced by the static semantics, while K-Java uses (a subset of) plain Java. ASM-Java is monolithic, while in K-Java the static and the dynamic semantics are two separate definitions that can be used independently. For example, other projects can use our static semantics to reduce the set of Java programs they need to handle.

JavaFAN \[19\] \[13\] is another large-scale executable semantics of Java, defined using term rewriting in Maude \[14\]. Our testing revealed that JavaFAN is overall less complete than ASM-Java, although JavaFAN passed a few tests that ASM-Java failed to pass. However, unlike ASM-Java, thanks to the
high-performance Maude model-checker \cite{16}, JavaFAN was successfully used to perform state-space exploration and model-checking of Java programs. For example, JavaFAN was able to detect the deadlock in the Dining Philosophers problem and prove the fixed program correct.

**Comparison.** Since both ASM-Java and JavaFAN are executable, we evaluated their completeness by running our comprehensive test suite (Chapter 8) with each. The results of our findings are presented in Figure 2.1. The list of Java features is divided into 10 large groups, separated by horizontal bars. The first 8 groups contain features introduced with Java 1.0: literals, expressions, statements, arrays, classes and instance members, interfaces, static members and static initialization, packages and identifier resolution. Group 9 includes features introduced with Java 1.2, and the last group includes the single new feature of Java 1.4 - `assert`.

Besides packages, ASM-Java does not define all literal forms, an important set of operators, the switch statement, array initializers, and complex cases of method overloading and method access modes. The remaining features of Java 1.0 are defined, except for some corner cases. JavaFAN, despite being more recent than ASM-Java, covers a smaller subset of Java. Yet, it surpasses ASM-Java in a few areas. JavaFAN supports a wider set of operators, but still not all of them, and it has better support for local variable shadowing. Yet many other features are not supported: switch and continue, advanced array features, many class-related features and interfaces. Also, JavaFAN makes no distinction between various integer types and integer overflow is not defined. From Java 1.2 and 1.4, it only supports the simplest feature, instance initializers.

**Methodology.** The level of granularity differs widely both among groups and among individual rows in Figure 2.1. We intentionally compressed large portions of Java into one row when all semantics defined them (e.g., Basic OOP). Yet, when we identified interesting features not supported by some of the semantics, we extracted them into individual rows. We were careful to interpret the test results objectively. In particular, we designed each test to touch as few Java features as possible other than the tested one, to minimize the number of reasons for failure (Chapter 8). In the rare cases when we were unable to identify why a particular test failed, we gave the semantics the benefit of the doubt (except for K-Java).
2.2 Other Large-Scale Executable Semantics

Several large-scale semantics have been defined recently. Here we only mention those which had a direct influence on K-Java. The first large-scale semantics developed in $K$ was that of C [18]. It covered all the C features, was tested using the GCC torture test suite [22], and was used to explore C undefinedness. A large fragment of PHP [20], also defined in $K$, was tested for conformance using the PHP standard test suite. The semantics was applied to prove simple properties using a $K$-based language-independent framework for symbolic execution [2] and model-checking. Two different semantics of large fragments of Python were defined using $K$ [28] and resp. PLT Redex [41]. Both were tested against publicly available test suites. The ASM-Java techniques were also successfully reused to define C# [9], benefiting from the fact that Java and C# are similar. JSCert [4] is an almost complete semantics for JavaScript in Coq, using a methodology that ensured a close visual resemblance to the standard, and tested against the JavaScript standard test suite.

A special place in this list is KJS [39], the first complete semantics of JavaScript (version ECMAScript 5 [15]), defined in $K$, published shortly after K-Java [8]. It defines not only the semantics, but also (in contrast to K-Java) the syntax of JavaScript, being the first large-scale semantics in $K$ that has the syntax incorporated in the definition. KJS was applied to deductively verify non-trivial programs, such as operations on AVL trees, using reachability logic [46]. This is the first to our knowledge demonstration of reachability logic on a complete real-world programming language, previous demonstrations [45] targeting a core subset of C [24]. K-Java was not used for deductive verification yet, being developed on an older version of $K$. The migration to the latest version of $K$ is underway at the moment of writing.
<table>
<thead>
<tr>
<th>Feature</th>
<th>AJ</th>
<th>JF</th>
<th>KJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic integer, boolean, String literals</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Other literals</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Overflow, distinction between integer types</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Prefix ++i --i, also += -= . . . ,</td>
<td></td>
<td>&amp;&amp;</td>
<td>○</td>
</tr>
<tr>
<td>Bit-related operators:</td>
<td>&amp; ^ &gt;&gt; &lt;&lt; &gt;&gt;&gt;</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Other integer operators</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>String + &lt;other types&gt;</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Reference operators</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Basic statements</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Switch</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Try-catch-finally</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Break</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Continue</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Array basics</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Array-related exceptions</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Array length</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Array polymorphism</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Array initializers</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Array default values</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Basic OOP - classes, inheritance, polymorphism</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Method overloading – distinct number of arguments</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Method overloading without argument conversion</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Method overloading with argument conversion</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Method access modes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Instance field initializers</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Chained constructor calls via this() and super()</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Keyword super</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Interfaces</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Interface fields</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Static methods and fields</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Accessing unqualified static fields</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Static initialization</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Static initialization trigger</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Packages</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Shadowing</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Hiding</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Instance initialization blocks</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Static inner classes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Instance inner classes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Local &amp; anonymous classes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Assert statement</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

Support level: ● = Full ○ = Partial ○ = None
AJ represents ASM-Java [48], JF is JavaFAN [19] and KJ is our work.

Figure 2.1: Completeness comparison of various semantics of Java
Chapter 3

Introduction to \(K\) Framework

In this chapter we first include a tutorial for \(K\), for readers new to \(K\) framework, that will help understanding the rest of the thesis (Section 3.1). Then we describe how \(K\) can be used to define semantics in Abstract Syntax Tree format (Section 3.2).

3.1 A \(K\) Tutorial

\(K\) is a framework for engineering language semantics. Given a syntax and a semantics of a language, \(K\) generates a parser, an interpreter, as well as formal analysis tools such as model checkers and deductive theorem provers, at no additional cost. It also supports various backends, such as Maude and, experimentally, Coq, that is, it can translate language semantics defined in \(K\) into Maude or Coq definitions. The interpreter allows the semanticists to continuously test their semantics, significantly increasing their effectiveness. Furthermore, the formal analysis tools facilitate formal reasoning for the given language semantics, which helps both in terms of applicability of the semantics and in terms of engineering the semantics itself; for example, the state-space exploration capability helps the language designer to cover all the non-deterministic behaviors of certain language constructs or combinations of them.

In this section we will present the main capabilities of \(K\) based on the semantics of a simple programming language – IMP. The content of this section is adapted from tutorials bundled with \(K\) distribution [23]. IMP is considered a folklore language, with no official inventor. It has been used in many papers and textbooks, often under another name or with slight syntactic variations. The language supports the most basic imperative programming constructs, specifically a few arithmetic an boolean expressions, variable assignment, if, while and block statements.
3.1.1 IMP Language Syntax

Syntax definitions in $\mathbb{K}$ use BNF notation. Under the hood, the version of $\mathbb{K}$ used for this thesis uses SDF \[30\] for parsing. This allows the syntax notation of $\mathbb{K}$ to extend BNF with a few attributes and notations inspired from SDF. In addition, there are several attributes specific to $\mathbb{K}$. All the attributes used in K-Java are described in this chapter. In this thesis (both in syntax and rule definitions) we represent terminals like this and non-terminals (also called syntactic sorts) $\text{LikeThis}$.

Below we give the syntax for IMP expressions and statements:

```
SYNTAX AExp ::= Int
    | Id
    | AExp + AExp [strict, klabel('Plus)]
    | AExp / AExp [strict, klabel('Div)]
    | ( AExp ) [bracket]

SYNTAX BExp ::= Bool
    | AExp <= AExp [seqstrict, klabel('LtEq)]
    | ! BExp [strict, klabel('Not)]
    | BExp && BExp [strict(1), klabel('And)]
    | ( BExp ) [bracket]

SYNTAX Block ::= { } [klabel('Block)]
    | { Stmt } [klabel('Block)]

SYNTAX Stmt ::= Block
    | Id = AExp ; [strict(2), klabel('Assign)]
    | if ( BExp ) Block else Block [strict(1), klabel('If')]
    | while ( BExp ) Block [klabel('While')]
    | Stmt Stmt [klabel('StmtSeq')]
```

The syntax for $\div$, $+$ and $!$ is annotated with attribute $\text{strict}$, meaning that the arguments of the respective syntax should be evaluated in any (fully interleaved) order, before the rules for the given syntactic production could apply. In real-life languages, nondeterministic order of evaluation was required, for example, to define the semantics of expressions in C \[18\], as the C standard allows some nondeterminism in subexpression evaluation.
for performance optimization purposes. The operator $\leq$ is defined to be `seqstrict`, which means that it evaluates its arguments in order, from left to right. Conversely, `seqstrict` was widely used in the present work, because Java Language Specification (JLS) requires all subexpressions of an expression to be evaluated left to right. The attribute `strict(1)` of the operator `&&` means the syntax is only strict in its first argument, because the rules for `&&` require only the first argument to be evaluated.

The attribute `bracket` for parentheses tells the parser to not produce a separate node for this production in the AST, but to inline its content instead. Thus, no semantics is required for parentheses. The attribute `klabel` will be explained at the end of this chapter.

An IMP program consists from a list of integer variable declarations, followed by a statement.

```
SYNTAX  Pgm ::= int Ids ; Stmt [klabel('Pgm)]
```

```
SYNTAX  Ids ::= List{Id , , } [klabel('Ids)]
```

This is the only place inside the program where variable definitions are allowed. The variables are initialized with 0. The second syntax definition above defines `Ids` as a syntactic list of elements `Id` separated by `","`. In $\mathbb{K}$, syntactic lists can also be empty. That is, it is possible to have an IMP program with no variables, like “`int; {}`”. Syntactic lists cannot be used as part of larger syntactic declarations, they should have their separate syntactic declaration with a separate sort (like the `Ids` above). Such lists are the $\mathbb{K}$ means to represent repetition (traditionally represented by `*` and `+` notation in Extended BNF).

Figure 3.1 contains an example IMP program computing the sum of numbers from 1 to (a hardcoded) $n$.

### 3.1.2 IMP Language Semantics

The first part of any language semantics in $\mathbb{K}$ is the basic semantic infrastructure, consisting from definitions for `values/results` and the `configuration`. 

20
int n, sum;
  n = 100;
  sum = 0;
  while (!(n <= 0)) {
    sum = sum + n;
    n = n + -1;
  }

Figure 3.1: An example IMP program, computing the sum from 1 to $n$

Values and results

In IMP, values (or computation results) may be either integers or booleans. For both forms, $\mathbb{K}$ builtin variants $\text{Int}$ and $\text{Bool}$ are used, but for real-life languages often custom-defined sorts are necessary.

\[
\text{SYNTAX} \quad K\text{Result} ::= \text{Int} \\
\quad \quad \quad \quad \quad | \quad \text{Bool}
\]

Next we will describe a few general facts about $\mathbb{K}$: the computations and the relationship between strictness and $\mathbb{K}$ results. Computations extend the abstract syntax of the defined language with a list structure, using $\rightsquigarrow$ (read “followed by” or “and then”) as a separator. $\mathbb{K}$ provides a builtin sort, $K$, for computations. The elements of this list may be any syntax definitions declared with \text{SYNTAX}. This sort allows $\mathbb{K}$ rules to uniformly operate over all syntax definitions in the semantics.

The computation structures of the form $t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots \rightsquigarrow t_n$ may intuitively be understood as a list of tasks to be processed in order. The computation is typically initialized with the original program as one task, which is then unfolded into a task sequence by the rules. The strictness attributes used in the syntax definitions are implemented through such a task unfolding. $\mathbb{K}$ compiles these attributes into rules. For example, the attribute \text{strict}(2) of the assignment statement above is compiled into the following two rules:

\[
\begin{align*}
X &= E \quad \Rightarrow \quad E \rightsquigarrow X = \Box \quad \text{REQUIRES} \quad E \notin K\text{Result} \\
E \rightsquigarrow X = \Box \quad \Rightarrow \quad X = E \quad \text{REQUIRES} \quad E \in K\text{Result}
\end{align*}
\]

The first rule above extracts $E$ out of its context into a separate task on top of computation in case it is not a $K\text{Result}$. Now other rules of the semantics could apply, until eventually $E$ is transformed into a computation result. At
this point the second rule from above matches, plugging the result back into its
context. The rules of the first type above are called heating rules and rules of
the second type cooling rules. Similar rules are generated for other arguments
affected by the strictness mechanism. When syntax definitions are strict only
in some of their arguments, the corresponding positions of the arguments
in which they are strict are explicitly stated in the argument of the strict
attribute, e.g., strict(2) like above, or strict(2, 3) for an operation strict
in its second and third arguments, etc. Syntax constructs simply declared as
strict are strict in all their arguments. Similarly, seqstrict attribute may
be declared with or without parameters. For both attributes, the arguments
are enumerated from left-to-right. The order is irrelevant for strict, but
it is relevant in case of seqstrict. For example, the heating/cooling rules
generated for the <= construct above, annotated with seqstrict, are:

\[
\begin{align*}
\text{E}_1 \leq \text{E}_2 & \Rightarrow \text{E}_1 \wedge \Box \leq \text{E}_2 \quad \text{REQUIRES} \quad \text{E}_1 \not\in K\text{Result} \\
\text{E}_1 \wedge \Box \leq \text{E}_2 & \Rightarrow \text{E}_1 \leq \text{E}_2 \quad \text{REQUIRES} \quad \text{E}_1 \in K\text{Result} \\
\text{E}_1 \leq \text{E}_2 & \Rightarrow \text{E}_2 \wedge \text{E}_1 \leq \Box \quad \text{REQUIRES} \quad \text{E}_1 \in K\text{Result} \land \text{E}_2 \not\in K\text{Result} \\
\text{E}_2 \wedge \text{E}_1 \leq \Box & \Rightarrow \text{E}_1 \leq \text{E}_2 \quad \text{REQUIRES} \quad \text{E}_1 \in K\text{Result} \land \text{E}_2 \in K\text{Result}
\end{align*}
\]

Note that in the rules above \text{E}_2 is heated for evaluation only when \text{E}_1
is already evaluated. These rules show how computation results are used
to trigger both heating and cooling rules at the right point of computation.
Computation results are language-dependent and their choice in complex
languages like Java is non-trivial. They cannot be automatically inferred, this
is why explicit definitions for \textit{KResult} are needed.

**Configuration**

A \texttt{K} configuration represents a (possibly nested) collection of cells. The
configuration of IMP contains only two cells: \texttt{\langle\langle\rangle\rangle}_k for the computation and
\texttt{\langle\langle\rangle\rangle}_\text{state} for the state. The two cells are placed inside another cell, the “top”
cell \texttt{\langle\langle\rangle\rangle}_T, for clarity.

**CONFIGURATION**

\[
\texttt{\langle\langle \texttt{\$PGM:Pgm} \rangle\rangle}_k \langle\langle \texttt{\_Map} \rangle\rangle_{\text{state}} \text{T}
\]

The notation inside the cells represents their initial state. Here \texttt{\$PGM}
is a special variable that represents the initial program. The “\texttt{\_Map}” in the
\(\text{state}\) is \(\mathbb{K}\)'s notation for an empty map. Technically, it is a constant which is the unit, or identity, of all maps in \(\mathbb{K}\). Similar dot units exist for all \(\mathbb{K}\) collections, such as lists, sets, multi-sets, etc.

The configuration serves as a backbone for the process of configuration abstraction (presented below) which allows mentioning only the relevant cells in each semantic rule, the rest of the configuration context being inferred automatically.

**Variable lookup**

The rule for variable lookup applies when the top of computation (e.g. the first computation item) is a program variable \(X\) of the sort \(Id\). The variable is matched in the \(\text{state}\) cell by the map entry notation of the form \(X \mapsto I\).

In \(\mathbb{K}\) notation, a vertical line represents a rewrite. The term above the line is the left-hand side (LHS), while the term below is the right-hand side (RHS). Every rule variable (like \(X\) or \(I\)) has a corresponding sort. The variable with its associated sort is represented by \textbf{Var:Sort} notation. This explicit sorting is optional; if a variable has no explicit sort specified, the widest sort compatible with all instances of that variable inside a rule is used. In most cases this implicit sort is \(K\).

In the rule above, if an entry \(X \mapsto I\) does not exist in \(\text{state}\), then the rewriting process will get stuck. This way the semantics of IMP disallows uses of uninitialized variables. Inside a cell, the notation “…” represents an arbitrary (possibly empty) content. Consequently, inside \(\text{state}\) the variable to be looked up is the first task, but the cell may contain other tasks as well (denoted by “…” at the end). Conversely, inside \(\text{state}\) the binding can be anywhere in the cell (the cell contains “…” at both sides). In \(\mathbb{K}\), rules can have names, such as \textbf{LOOKUP} for the rule above. Rule names are mainly used for documentation purposes, to increase the clarity.
Arithmetic operators

RULE

\[
\frac{I_1: \text{Int} + I_2: \text{Int}}{I_1 +_{\text{Int}} I_2}
\]

RULE

\[
\frac{I_1: \text{Int} / I_2: \text{Int}}{I_1 \div_{\text{Int}} I_2}
\]

REQUIRES \( I_2 \neq/= \text{Int} \ 0 \)

The rules of this form, that do not specify any cells, are only matched at the beginning of the \( \langle \rangle_k \) cell, as this is the expected behavior in most cases. For example, the rule for \( _+_{\text{Int}} \) operator will be expanded by the configuration abstraction mechanism into:

RULE

\[
\left\langle \left\langle \frac{I_1: \text{Int} + I_2: \text{Int}}{I_1 +_{\text{Int}} I_2} \right\rangle_k \left\langle \rho \right\rangle_{\text{state}} \right\rangle T
\]

The configuration abstraction mechanism might not be obvious for IMP, but it is essential for modularity for a large definition with dozens of cells like K-Java.

The operations with subscripts in the right-hand sides of the rules above are builtin (provided by K), associated with the corresponding builtin sort. Note that the variables appearing in these rules have integer sort. This ensures the rules above will only apply when the arguments of the arithmetic constructs are evaluated into final results by the strictness mechanism. We chose not to label these rules, for demonstration purposes.

Sometimes it is required for a particular rule to match in any context. This is achieved by tagging the respective rules with the attribute \texttt{anywhere}. Another possibility is to apply \texttt{function} attribute to syntax definitions of some auxiliary constructs — all the rules defining such 'pure functions' act like \texttt{anywhere} rules. In addition, pure functions are allowed to be used inside side-conditions of rules.
Boolean expressions

In the rules below, note the 'lazy semantics' for the second argument of $\&\&$. It is evaluated only if the first argument evaluates to true. This is the reason why $\&\&$ syntax was annotated strict only in its first attribute. In the second rule for $\&\&$ below, the second argument is not used anywhere in the RHS, thus its name is irrelevant. $\star$ allows in such cases using anonymous variables, represented as “—”.

RULE

$$
\begin{align*}
\text{I1:} \text{Int} & \leq \text{I2:} \text{Int} \\
\text{I1} & \leq_{\text{Int}} \text{I2}
\end{align*}
$$

RULE

$$
\begin{align*}
\text{! } \text{T:} \text{Bool} \\
\neg_{\text{Bool}} \text{T}
\end{align*}
$$

RULE

$$
\begin{align*}
\text{true} \ & \& \ B \\
B
\end{align*}
$$

RULE

$$
\begin{align*}
\text{false} \ & \& \ — \\
\text{false}
\end{align*}
$$

Blocks

The empty block $\{\}$ is simply dissolved. The $\star$ below is the unit of the computation list structure $\star$, that is, the empty task.

RULE

$$
\begin{align*}
\{ \} \\
\star
\end{align*}
$$

Similarly, the non-empty blocks are dissolved and replaced by their state-
ment contents, thus effectively giving them a bracket semantics; we can afford
to do this because IMP does not have block-local variable declarations.

RULE

\[
\{ \; S \; \} \\
S
\]

Assignment

The assigned variable is matched in the \( \{ \)state\, where its value is rewritten
into the new value. Inside \( \} \)k\, the assignment is dissolved.

RULE ASSIGNMENT

\[
\langle \begin{array}{c}
\text{X} = \text{I: Int } ; \\
\text{} \end{array} \rangle_k \quad \langle \begin{array}{c}
\text{...} \\
\text{X} \mapsto \text{I } \end{array} \rangle \; \text{state}
\]

Sequential composition

Sequential composition is implemented through translation into \( \mathbb{K} \)'s builtin
task sequentialization operation.

RULE SEQUENTIAL

\[
S_1 \; S_2 \\
S_1 \; \bowtie \; S_2
\]

If statement

The if statement has two rules, one for the case when the condition evaluates
to true, the other one for false. The rules rewrite the if into its second or
third argument, respectively. Recall that if was declared strict only in its
first argument, thus the other arguments are not altered by strictness.

RULE IF-TRUE

\[
\text{if (true) } S \; \text{else } — \\
S
\]
RULE IF-FALSE

\[
\text{if (false) } \rightarrow \text{ else } S
\]
\[
S
\]

While loop

The semantics of the while loop is given by unrolling.

RULE WHILE

\[
\text{while (B) S}
\]
\[
\text{if (B) } \{ \text{ S while (B) S } \} \text{ else } \{ \}
\]

Programs

When the execution of an IMP program begins, first the variable declarations are processed and variables are initialized in the \( \{ \text{store} \) with the value 0. Then, the statement representing the program body is executed.

The processing of variable declarations requires two rules: one when there are unprocessed variables left, another one if all the variables have been processed. The first rule initializes the first variable in the list with 0, and deletes it from the declaration. The side condition checks that the variable have not been declared already.

RULE PROGRAM-STILL-VARS

\[
\begin{align*}
\text{int} & \quad \text{X:Id}, \quad \text{Xs:Ids} \\
\quad & \quad \text{Xs}
\end{align*}
\]
\[
\text{;} \
\]
\[
\text{\{ } \quad \text{\rho:Map} \
\]
\[
\quad \text{\{ } \quad \text{\'Map} \\
\]
\[
\quad \text{\{ } \quad \text{\{ } \quad \text{X} \mapsto 0 \\
\]
\[
\text{\{ } \quad \text{\} } \quad \text{\{ } \quad \text{\} }
\]
\[
\text{state}
\]

REQUIRES \( \neg_{\text{Bool}} (X \text{ in keys(\{ \rho \})}) \)

Note that the rule above matches the entire \( \{ \text{store} \) cell, as at the beginning of execution \text{int;}\_\_ is the only term in the computation. The anonymous variable here matches the program body.

The second rule matches after all variable declarations have been processed (and thus dissolved). It dissolves the empty variables declaration, leaving the statement \( S \) to be evaluated.
**Modularity.** One of the most appealing aspects of $\mathcal{K}$ is its modularity. It is very rarely the case that one needs to touch existing rules in order to add a new feature to the language. There are two $\mathcal{K}$ features specifically targeting modularity. First, the configuration can be structured as nested cells. Second, the language designer needs only to mention the cells that are needed in each rule, and only the needed portions of those cells. This modularity contributes not only to compact and thus human readable semantics, but also to the overall effectiveness of the semantics development process. For example, if a new cell is added to the configuration later on in order to support a new feature, rules that don’t use it don’t have to change. Modularity was only touched by the rules of IMP, but is used in its full strength in K-Java. Without modularity, defining a complete language of this scale would not be possible.

Some additional features of $\mathcal{K}$ will be presented on a by-need basis in the following chapters. The reader interested in finding more about $\mathcal{K}$ is invited to check the paper [43], and $\mathcal{K}$ tutorials [23].

### 3.2 AST-Based Terms in $\mathcal{K}$

**Definition.** An abstract syntax tree (AST) is a representation of the information obtained by parsing a program, which omits elements such as parentheses and semicolons, syntactically important but having no semantic value. An AST contains a node for each semantically meaningful token identified during the parse.

Both programs and rules are represented internally by $\mathcal{K}$ tools as AST. The language developer has some control over the AST-representation through syntax attributes `klabel` and `bracket` (see syntax definitions in Section [3.1.1]). The attribute `klabel` tells $\mathcal{K}$ what label should be used for the AST nodes corresponding to a particular syntax production. The attribute `bracket` instructs $\mathcal{K}$ to not generate an AST node for the annotated production. The AST representation of the program in Figure [3.1] is given in Figure [3.2].
'Pgm(
  'Ids('Id(n), 'Ids('Id(sum))),
  'StmtSeq(
    'StmtSeq(
      'Assign('Id(n), 100),
      'Assign('Id(sum), 0)
    ),
    'While(
      'Not('LtEq('Id(n), 0)),
      'Block(
        'StmtSeq(
          'Assign('Id(sum), 'Plus('Id(sum), 'Id(n)));
          'Assign('Id(n), 'Plus('Id(n), -1))
        )
      )
    )
  )
)

Figure 3.2: The AST representation of the IMP program from Figure 3.1

AST-based terms may contain rule variables, and may be freely used inside \( \mathcal{K} \) rules. For example, the rule for + may be written in the following way:

\[
\text{RULE}
\]

\[
'Plus(\mathbf{I1:}Int, \mathbf{I2:}Int)
\]

\[
\mathbf{I1} +_{\text{int}} \mathbf{I2}
\]

The rule above is equivalent to the original one defined over syntax terms.

AST-based terms have a number of valuable uses. One of them is solving parsing ambiguities within rules. Because the syntax for a rule term is a combination of the defined language syntax and \( \mathcal{K} \) notation, rules sometimes cannot be unambiguously parsed. Writing the ambiguous part in AST form generally solves the problem.

Another, more valuable use of AST form is the ability to handle the language syntax and semantics independently, by different tools. For many real-world languages, there are ready-available parsers developed outside \( \mathcal{K} \) "ecosystem". By connecting \( \mathcal{K} \) to such an external parser, the syntax designer
may completely eliminate language syntax from his definition, and let K
only handle the semantics. In this case all the rules have to be defined in
AST-based form. This approach was used to define many languages, such as
C [18], PHP [20], as well as the present work, K-Java [8].
Chapter 4

Parsing Java Programs

The content of this chapter is partially based on [7].

K framework has a builtin support to define the syntax and to parse programs. However, when K-Java project started, the syntactic capabilities of K (then in its 2-nd version) were quite limited. Those capabilities were ok to define the syntax of toy languages like IMP (Section 3.1) or the object-oriented KOOL [32], but not for real-world ones. Version 2 of K had very little support for handling ambiguities, and in fact, toy languages mentioned above had at that time a syntax specially crafted to walk around this problem. Moreover, even the current version of K only supports standard BNF syntax style, while many grammars would benefit from Extended BNF format.

Since the primary focus of K Framework is the semantics and not the syntax, it supports connection to an external parser. Several existing language definitions in K use either parsers generated by a standard tool like ANTLR [40], or custom-made parsers. Either way, crafting a parser compatible with K Framework requires a significant amount of work. For this task it would be advantageous to have a tool capable to generate the right parser out of the box, based on the syntax definition alone.

For Java, such a syntax definition was readily available: Java-Front [12], a complete syntax for Java 5, defined in SDF (Syntax Definition Formalism) [31], which is a mature and widely used tool. The fact that K-Java only supports Java 1.4 was not an impediment, since Java 5 syntax is a strict superset of Java 1.4 syntax.

SDF was also convenient in terms of tool interface. For a given grammar, SDF generates a command-line tool that outputs the AST of the input program to the standard output. Many other parser generators produce the parser source code instead, which requires laborious customizations to make use of the AST, even if simply printing it. Yet the format of the SDF-produced AST, called ATerm [11], is slightly different than the format expected by K.
Pgm(
    Ids([Id(n), Id(sum)]),
    StmtSeq([
        StmtSeq([
            Assign(Id(n), 100),
            Assign(Id(sum), 0)
        ]),
        While(
            Not(LtEq(Id(n), 0)),
            Block(
                StmtSeq([[
                    Assign(Id(sum), Plus(Id(sum), Id(n))),
                    Assign(Id(n), Plus(Id(n), -1))
                ])
            )
        ])
    ])
)

Figure 4.1: The ATerm representation of the IMP program from Figure 3.1

For example, the ATerm representation of the program from Figure 3.1 is given in 4.1 As you can see, the difference from K AST (Figure 3.2) is quite small. All that was remaining for us was to develop a transformation that converts the ATerm into K AST format. We presented both the transformation tool and the methodology for defining a semantics in K using the tool in [7], which we let the reader to check for details.

When K is used with an external parser, the semantics developer is free to define a part of the language syntax in K as well, or to not define any syntax at all. For the part that have no corresponding syntax inside K definition, the semantic rules have to be in the AST form. This is exactly how K-Java was developed for most of its time. Only when the semantics was complete, we re-wrote a small part of K-Java rules to Java syntax form, in order to better present them in our main paper ([8]) and this thesis. While all the rules presented in the main part of the thesis are defined over Java syntax, the rules in the appendixes remained mostly in the AST format.

After our work on K-Java was complete, the project continued to develop. An ongoing effort is to define the entire syntax of Java as integral part of K-Java, with the goal to get rid of the external parser. This is done by
Shijiao Yuwen from Formal Systems Laboratory (FSL) of UIUC. The syntax is actually an adaptation from Java-Front. With the kind permission from FSL we included the syntax Shijiao developed into this thesis as Appendix D. Although the syntax is fairly complete, it still contains some errors that prevents its use for parsing programs. Nevertheless, the syntax is useful as a reference. It also defines all the $\texttt{klabel}$ attributes used inside the rules. The effort of converting K-Java rules to the syntactic form is also in progress.

The ATerm to $\mathbb{K}$ AST transformer proved to be useful outside K-Java. The PHP semantics in $\mathbb{K}$ 20 adopted the transformer and the methodology presented by us in 7.
Chapter 5
Static Semantics

K-Java is divided into two separate definitions: static semantics (covered in this chapter) and dynamic semantics (the next chapter). An overview of the whole K-Java is presented in Figure 5.1.

The static semantics takes as input the AST representation of a Java program and produces a preprocessed program as the output. It performs computations that could be done statically, and are referred in JLS as compile-time operations. Such computations include converting each simple class name into a fully qualified class name or computing the static type of each expression. The preprocessed AST is then passed to the dynamic semantics for execution.

We choose to present static K-Java from two complementing perspectives. First, the functionality is illustrated as a set of transformations over programs (Section 5.1). Next, the inner workings are presented as a sequence of phases (Section 5.2), detailing how the configuration content changes during each phase. We choose not to present any rules in this chapter, as the static semantics uses a large number of inter-dependent auxiliary constructs (read: is not very modular:) ) that makes it difficult to extract a reasonably sized self-contained fragment. For the reader interested to see all the details, the entire static semantics is included in Appendixes B and C.

5.1 Transformations

The role of static semantics is to transform the program in a way more suitable for execution, by computing some of the information required for execution statically. This can be accomplished in multiple ways. For example, in ASM-Java[48] and ASM-C#[9] the output of the static semantics is an annotated program. We choose to be more restrictive, confining the output of the static semantics to valid Java programs, without any annotations.
Figure 5.1: The high-level structure of K-Java
Figure 5.2: Transformations performed by the static semantics
The preprocessed program contains just a subset of features of the original program, leading to a reduced number of cases that has to be supported by the dynamic semantics. Naturally, such preprocessed programs are expected to also be equivalent to the original program. The choice brought us both advantages and limitations; we discuss them at the end of the chapter.

Being a Java to Java transformation, the effect of static K-Java is best illustrated on example programs. Figure 5.2 covers the most important individual transformations. The left column contains sample input fragments together with the minimal relevant context. The right column represents the transformed fragment.

The simple class names are transformed into fully qualified class names (row 1). This allows us to eliminate imports clauses in preprocessed programs. Each expression is additionally wrapped into a cast expression that encodes its static type (row 2). During execution this cast is treated like a regular cast. The example in the figure also illustrates binary numeric promotion: the result type of the addition between an `int` and a `long` is a `long`.

In addition, for method calls each argument is wrapped into one more cast, namely to the expected parameter type (row 3). This allows us to effectively encode method overloading.

Row 4 illustrates the preprocessing of fields accessed by simple names. Each such field access is converted into a qualified access. If the field is non-static, it will be qualified by `this` (first case in row 4), and `this` should be additionally cast to the current class type, according to rule 2. If the field is static, it will be qualified with the class that defined the field (second case). The same transformation is performed for methods called without a qualifier.

This last transformation allows us to disambiguate among the four roles of a name, namely package, class, field or local variable:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Syntactic Context</th>
<th>Resulting Role</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Id</code></td>
<td>package</td>
<td>package</td>
</tr>
<tr>
<td><code>Id</code></td>
<td>package or class</td>
<td>package</td>
</tr>
<tr>
<td><code>Id</code></td>
<td>class</td>
<td>class</td>
</tr>
<tr>
<td><code>((Type) Qual.Id)</code></td>
<td>expression</td>
<td>field</td>
</tr>
<tr>
<td><code>((Type) Id)</code></td>
<td>expression</td>
<td>local var</td>
</tr>
</tbody>
</table>

An identifier inside a syntactic context where only a package is allowed will be considered a package. If either a package or a class is allowed, then
the identifier will also be considered a package (here we consider classes that
are used as qualifiers to refer to an inner class also as packages). A field or
a local variable will always be part of a cast expression, thus will always be
inside an expression context. If the respective identifier is qualified, then it is
a field. Otherwise it is a local variable.

Continuing with row 5, if a constructor does not call another constructor
on the first line, then it should call the base class constructor, super(),
except if the current class is Object. For legibility, we omit the already dis-
cussed transformations (e.g., wrapping into cast expressions) in the remaining
examples.

Initializers of static fields as well as static initializing blocks are collected
into one big static initializing block (row 6), which is invoked during execution
when the class is for the first time accessed by the program. For instance
initializers the same procedure cannot be followed (row 7). Instead, all the
instance initialization code is collected into the method $init(), that is called
by the constructors right after the base constructor super() was called.

Arguably the most complex transformation is that of local classes. It
is encoded into a separate module with 26 rules, one of the largest of K-
Java. Local classes are converted into instance inner classes within the same
enclosing class. The unusual difficulty of this transformation arises from the
ability of local classes to access final local variables visible at the point when
the class was declared. The most common case is presented in row 8. For
each such local class L, an additional inner class LEnv is defined that stores
the enclosing local environment of L. LEnv contains a field for each variable
in the environment. Accesses to the local environment of L are encoded as
accesses to fields of LEnv.

Finally, anonymous classes are first converted into local classes, which are
in turn converted into inner classes (the last row).

5.2 Phases

The static semantics consists of several phases, depicted in Figure 5.1. Each
phase digs deeper into the syntactic structure of Java and either performs a
set of transformations over the program or computes some new configuration
cells.
Figure 5.3: The static semantics configuration
Figure 5.3 contains the whole configuration of the static semantics. The cells may be divided into two categories: top-level cells residing directly under \( T \) except \( \text{classes} \), and \( \text{classes} \). The cell \( \text{classes} \) is a collection of \( \text{class} \) cells (multiplicity is denoted by * after the cell name). After the input program is fully processed, there is one \( \text{class} \) for each class or interface, both for JDK classes and for classes defined by the developer. Also, since Java supports nesting of classes, both top-level and nested classes have their separate \( \text{class} \). Inside each \( \text{class} \) there is a separate cell for every relevant information associated with a class — such as extends/implements clauses, imports, fields, methods, etc.

In the remaining of this section we present the evolution of the configuration through each phase, documenting alongside all the cells in the order in which they are first used. This way we illustrate the high-level mechanism by which the transformations in Figure 5.2 were achieved. For each phase we also give the reference to the appendix sections where phase definition may be found.

**Initial state** In the initial state all cells except the first three are empty (\( \text{classes} \) contains an empty collection). The first three cells have the following function:

- \( K_k \) — Holds the current computation. Is initialized with the AST representation of the program.
- \( K_{\text{program}} \) — A backup of program AST. Required because the initial AST is required in both 1st and 2nd phase of static K-Java, but is consumed by the first phase.
- \( \text{globalPhase} \) — The current global phase.

Our cell notation in this section includes for each cell the sort of its content. For example \( \text{globalPhase} \) has the sort \( \text{GlobalPhase} \), a sort that enumerates all the global phases. When we don’t want to restrict the content of a cell, it has the sort \( K \).

**Collect class names phase.** In this phase the initial program is traversed and all class names are collected into a map. The names map serves two purposes at later phases: to resolve simple class names into fully qualified
class names and to determine the set of classes available inside a package. Each class is traversed recursively, so that the inner classes are also registered. Yet the traversing does not proceed inside code blocks, thus classes defined at the block level (local and anonymous) are not discovered at this stage.

This phase computes one top-level cell:

- \( \langle \langle \langle \text{Map[PackageId} \mapsto \text{Map[Id} \mapsto \text{ClassType}] \rangle \rangle \rangle \text{namesToClasses} \) — A two-level map. First level is a map from package names to second-level maps. Second-level maps are from simple class names to fully qualified class names within the package. This cell is extensively used through the semantics. For inner classes, their enclosing package is the fully qualified class name of the directly enclosing class.

All the remaining global cells in Figure 5.3 are temporary cells associated with some particular phases. We will not cover them in this section. Their documentation is still provided in Appendix C.

Also during Collect class names the classes are first registered. In a newly created \( \langle \rangle \text{class} \) just a few sub-cells are initialized with data:

- \( \langle \text{ClassType}\rangle \text{classType} \) — The fully qualified class name. The identifier of the class.

- \( \langle \text{ClassMetaType}\rangle \text{classMetaType} \) — Represents whether the type stored in this cell is class or interface. To avoid terminology superfluousness, we will refer hereafter to both classes and interfaces as *classes*, making the distinction only when necessary.

- \( \langle \text{ClassAccessMode}\rangle \text{classAccessMode} \) — The access modifier of the class, either public or package.

- \( \langle \text{ClassPhase}\rangle \text{classPhase} \) — Represents the state of this class. In addition to the global computation phase, each class has its own lifecycle phase. This class phase allows tracing the degree to which each class was processed, and allows a higher flexibility of the order in which classes are processed at later phases. The class phases are:
  
  - Discovered — The initial phase. At the end of Collect Class Names all classes are in the state "Discovered".
- Stored
- Bases Resolved
- Declaration Processed
- Members Processed
- Folded

**Process compilation units phase.** In this phase the traversal starts again from the initial program AST (restored from \( \langle \text{program} \rangle \)) and performs two tasks. The first is to move class content from the initial AST to the \( \langle \text{class} \rangle \) cells inside the configuration, from where it will be easier to access during subsequent phases. From this point on the program is no longer a monolithic AST but a collection of \( \langle \text{class} \rangle \) cells. The granularity of the initial \( \langle \text{class} \rangle \) content is very rough. At this stage just the class declaration, including extends/implements clauses, is analyzed, and not the class members. The class content is analyzed recursively like in the the previous phase, to process all the inner classes. The second task of this phase is to process imports and produce the map of class names accessible for each top-level class.

The following new cells are filled in inside each \( \langle \text{class} \rangle \):

- \( \langle \text{ClassType} \rangle \_\text{enclosingClass} \) — The directly enclosing class, for inner classes, or no value for top-level classes.
- \( \langle K \rangle \_\text{rawExtends} \) — The extends clause of this class, in its raw (AST) form.
- \( \langle K \rangle \_\text{rawImplements} \) — The implements clause, in AST form.
- \( \langle K \rangle \_\text{rawDeclarations} \) — The class body, in AST form.
- \( \langle \text{Map}[\text{Id} \rightarrow \text{ClassType}] \rangle \_\text{cuImports} \) — A map from names accessible inside this class to fully qualified class names they represent. Only computed for top-level classes at this phase. For inner classes this cell remains empty.
- \( \langle \text{ContextType} \rangle \_\text{classContextType} \) — Either static or instance, for inner classes. Always static for top-level classes.
The class phase changes from Discovered to Stored. As we can see, the
cells computed so far contain all the data of the original program. Thus,
initial AST representation of the program is no longer needed.

**Process class declarations phase.** In this phase the simple class names
inside `extends/`implements clauses of each class are converted into fully
qualified class names. This allows us to compute the map of accessible class
names for inner classes. JLS has complex rules for the names accessible inside
inner classes that do not allow computing those names at an earlier stage.

During Process class declarations each class advances through two more
class phases: Bases Processed and Declarations Processed. First, for each class
the semantics attempts to resolve its `extends/`implements clauses into fully
qualified class names. The order in which dependencies are resolved depends
on both class inheritance relationships as well as nesting relationships. Once
the dependencies of a class are resolved, they are stored into a temporary cell:

- `{K}_{unprocessedBases}` — Initialized with the list of fully qualified class
  names for classes mentioned in `extends/`implements clauses of this class.

Once the content of `{K}_{unprocessedBases}` is created, the class enters into
Bases Processed phase. It then waits in this phase until all classes referred in
`extends/`implements reach the phase Declarations Processed. The restrictions
in JLS related to class dependencies guarantee that classes cannot have cyclic
dependencies, thus a class cannot get locked in the waiting state. The cell
`{K}_{unprocessedBases}` is used to determine the moment when the class may exit
the waiting state. Once a class reaches the phase Declarations Processed,
is is deleted from cells `{K}_{unprocessedBases}` of other classes. Thus, when all
`extends/`implements dependencies of a class reach the phase Declarations
Processed, the content of the its `{K}_{unprocessedBases}` cell becomes empty. Once
in this state, the class enters into the phase Declarations Processed itself and
computes three more cells:

- `{ClassType}_{extends}` — The base class, fully qualified.

- `{Set[ClassType]}_{implements}` — The list of directly implemented inter-
  faces, fully qualified.
• \( \langle \text{Map}[\text{Id} \mapsto \text{ClassType}] \rangle_{\text{imports}} \) — The map of classes accessible by simple name within the body of this class. The rules for computing this map are complex and include the following sources:

  - Imports declarations of the current compilation unit.
  - Classes declared within the package of the current compilation unit.
  - Classes accessible within the body of the directly enclosing class, if the current class is inner class.
  - Inner classes inherited from base classes, e.g. from extends/implements clauses.
  - Inner classes of this class itself.

The need to cover all these cases leads to the intricate order in which class dependencies have to be be resolved.

When a class enters the phase Declarations Processed, the cells \( \langle \rangle_{\text{rawExtends}} \), \( \langle \rangle_{\text{rawImplements}} \) and \( \langle \rangle_{\text{unprocessedBases}} \) are no longer needed and are discarded. Once all classes reach this phase the computation proceeds to the next global phase.

**Process class members phase.** At this stage the body of each class is processed and each member is stored in an appropriate cell. For methods, the types for its parameters and return value are resolved.

Each member is extracted from \( \langle \rangle_{\text{rawDeclarations}} \) of its class and registered into a more specific cell. Class members at this point could be one of the following: field, method, constructor, static initializer or instance initializer.

The following new class cells are produced:

- \( \langle \text{Set}[\text{ClassType}] \rangle_{\text{implTrans}} \) — The transitive closure of implemented interfaces. In the remaining phases this set is used by the subtyping relationship.

- \( \langle \text{Map}[\text{Signature} \mapsto \text{ClassType}] \rangle_{\text{methods}} \) — The map of accessible methods. Keys are method signatures, values are classes where methods are defined. Includes both methods declared within this class as well as methods inherited from base classes/ base interfaces.
• $\langle Bag \rangle_{md}$ — The collection of method declarations ($\langle methodDec \rangle$ cells) in the current class. This cell contains only a subset of methods from $\langle methods \rangle$, as the set of accessible methods from $\langle methods \rangle$ also includes methods inherited from base classes/interfaces. Hence the need for two separate collections. Each $\langle Bag \rangle_{methodDec}$ contains the following data:

  - $\langle Signature \rangle_{methodSignature}$ — The method signature, acting as identifier of the $\langle methodDec \rangle$.
  - $\langle Type \rangle_{methodReturnType}$ — The method return type.
  - $\langle List[ Param / ] \rangle_{methodParams}$ — The method parameters.
  - $\langle K \rangle_{methodConstrFirstLine}$ — The first line of a constructor (if this method is indeed a constructor, for classes other than Object). It contains a call to another constructor: either super() or this().
  - $\langle K \rangle_{methodBody}$ — The method body.
  - $\langle AccessMode \rangle_{methodAccessMode}$ — The method access mode.
  - $\langle ContextType \rangle_{methodContextType}$ — May be either static or instance.
  - $\langle MethodMetaType \rangle_{methodMetaType}$ — May be either method or constructor.

• $\langle K \rangle_{instanceFields}$ — The list of instance field declarations, stored as a list of local variable declaration statements, without initializers. Used during object instantiation.

• $\langle K \rangle_{instanceInit}$ — The list of instance initializers of the class combined into one big instance initializer. Instance field initializers are also concatenated into this cell in their textual order.

• $\langle K \rangle_{staticFields}$ — The list of static field declarations, in a similar format to $\langle instanceFields \rangle$.

• $\langle K \rangle_{staticInit}$ — The list of static initializers and static field initializers concatenated into one block.

• $\langle Map[Id \mapsto Value] \rangle_{constantEnv}$ — The map from compile-time constants to their actual values. Constants in Java have a slightly different semantics compared to final static fields. In particular, accessing them
don’t trigger static initialization of the declaring class. In order to expose this difference they have to be registered separately from static fields.

Once all the cells above are computed the class proceeds into the phase Members Processed (the last phase) and the cell \( \langle \rangle_{\text{rawDeclarations}} \) is deleted.

**Elaboration phase.** This phase involves analyzing the code blocks and is the most complex of all phases. The following transformations are performed:

- Each name is resolved into local variable, field, method, class or package. While a method may be distinguished from other categories purely syntactically, resolving to other categories requires knowledge of the names existing in the current context.

- Simple class names are resolved into fully qualified class names. Hereafter all the class names in the code are fully qualified.

- The compile-time type of each expression is inferred. Thus, when the code reaches execution phase, expressions are no longer in their initial form. Each expression is wrapped into a cast expression that encodes its compile-time type.

- For each method call the precise signature in inferred. Each method call argument is wrapped into a cast expression that encodes its expected type.

- Also at this stage block-level classes (local and anonymous) are first detected. Upon their detection, block-level classes must be passed through all the class phases, to be normalized to the same format as the other classes.

This phase introduces one temporary cell:

- \( \langle \langle M a p[ I d \rightarrow Type]\rangle_{\text{enclosingLocalEnv}} \) — The map from local variables of the current block to their types. Used during the processing of local classes.

No cells are consumed during elaboration. Instead, the code inside \( \langle \rangle_{\text{methodDec}}, \langle \rangle_{\text{instanceInit}}, \langle \rangle_{\text{staticInit}} \) is processed and stored back into the same cells.
Motivation for elaboration phase. Initially, K-Java was developed without an elaboration phase. For example expression types were computed along their value each time they were executed. Similarly, other computations that are now done in the elaboration phase were performed on the fly during execution. This approach worked up to a point, and produced a monolithic semantics which was more compact than the present one. However, when we reached the point to define the semantics of the conditional expression `?::` and of the local classes, it eventually became clear that we had to compute the static types upfront. As described in Section 1.3, it is impossible to compute the type of a conditional expression by evaluating its operands dynamically. Also, local classes had to be discovered and preprocessed prior to execution, in order to cover all the corner cases. For this we needed a pre-execution phase capable to dig into code blocks.

Folding phase. The last and simplest phase of preprocessing, Folding, combines the information stored in ⟨⟨⟨⟩⟩⟩ class cells back into one big AST. First the content of each class is assembled into ⟨⟨⟨⟩⟩⟩ of the respective class, and the class phase changes to Folded. Second, the AST representation of each class is appended into ⟨⟨⟨⟩⟩⟩ program. When this phase ends, the content of ⟨⟨⟨⟩⟩⟩ program is printed to the standard output.

This phase acts as an interface to the dynamic semantics. Since all the transformations performed by the static semantics maintain the validity of the program, the resulting preprocessed AST represents a valid Java program as well. This phase is technically unnecessary for the dynamic semantics alone, because Unfolding phase in dynamic K-Java (Figure 5.1) is the exact mirror of Folding, distributing the AST back into the same cells that were used to produce it. Folding and Unfolding phases were required to complete the division between the static and the dynamic parts of K-Java.

The preprocessed AST. The output of the static semantics is the preprocessed AST. This AST has a valuable property: it corresponds to a valid Java program, equivalent to the initial one. More precisely, for every Java program $P_1$ there is a program $P_2$ such that the preprocessed AST of $P_1$ is equal to the AST of $P_2$. We do not prove this property, we only state it, supported by the observation that every transformation performed by the static semantics preserves program equivalence.
Maintaining this equivalence imposed some challenges. First, we were forced to use only transformations that preserve program validity. All the transformations presented in Figure 5.2 are also performed by the Java compiler (javac), with the difference that javac produces bytecode. In contrast, not all transformations performed by javac were accessible to static K-Java. For example, javac flattens inner classes into top-level classes. We could not do the same because we would lose method access modes, and access modes have to be preserved (Section 1.3). Also, we could not use any auxiliary constructs in the preprocessed AST. Despite the challenges, we believe producing a preprocessed AST with stated properties was worthwhile, since it enables our static semantics to be usable in conjunction with tools outside the K ecosystem.
Chapter 6
Dynamic Semantics

This chapter contains the actual definition, using \( K \) rules, of a wide portion of dynamic K-Java. Section 6.1 illustrates the full configuration of dynamic K-Java. Key auxiliary notation used through this chapter is presented in Section 6.2. We continue with the semantics of expressions, giving the definition of a few numeric and a few reference expression in Section 6.3. The Section 6.4 gives the semantics of selected statements: if, block, while and exception handling. The Section 6.5 presents the portion of the configuration representing the memory model. It is then used to present rules for variable access for various kinds of variables in Section 6.6. The instantiation of new objects is defined in Section 6.7. Among the most complex parts of dynamic K-Java is method invocation, presented in Section 6.8. The final section 6.9 gives the semantics for multithreading constructs: thread creation, synchronization and wait/notify mechanism.

A part of the work in this section is from [8] and [6]. The full dynamic K-Java semantics, along with some extra documentation, is included in Appendixes A and B, although rules included there are in AST format.

6.1 Configuration

The entire configuration of dynamic K-Java is shown in Figure 6.1. For each leaf cell, the notation inside represents the sort of the cell content, with most of the sorts being explained in the following sections. Here we give just a high-level overview, while the rest of the sections include a detailed explanation for most of the cells, alongside the rules that use them.

The configuration consists of three large cell groups: collection \( \{\}\text{threads} \) of cells \( \{\}\text{thread} \), collection \( \{\}\text{classes} \) of cells \( \{\}\text{class} \), and the remaining cells directly under \( \{\}\text{T} \), which we call global cells.

Multithreading in K-Java is supported by having multiple cells \( \{\}\text{thread} \),
Figure 6.1: The dynamic semantics configuration
each being a set of cells representing the thread context. This include
the current computation \( \langle k \rangle \), the method call stack \( \langle \text{stack} \rangle \), and various other
cells related to the currently executed method inside \( \langle \text{methodContext} \rangle \). The
lower row inside \( \langle \text{thread} \rangle \) contains data required for communication between
threads.

The static class-specific information, and the dynamic information inde-
dependent of actual objects, is held within \( \langle \text{class} \rangle \) cells. The subcells holding
static information (e.g. unaltered during execution) have the same name and
content as their counterparts in the static semantics.

The global cells include program startup data (the first row), cells related
to memory model (from \( \langle \text{store} \rangle \) to \( \langle \text{objectStore} \rangle \)), standard input \( \langle \text{in} \rangle \) and
output \( \langle \text{out} \rangle \), and cells related to multithreading (the bottom two rows).

Compared to the static semantics configuration (Figure 5.3), here the
majority of the cells specific to preprocessing do not appear (the exception
being cells for unfolding phase). Instead, we have new cells supporting
execution-related functions: multithreading, method invocation, memory
model and standard input/output.

**Dynamic Semantics Phases**
The dynamic semantics has just two phases:
Unfolding and Execution (Figure 5.1), identified by the state of \( \langle \text{globalPhase} \rangle \).
In the unfolding phase the AST representation of the program is extracted from
\( \langle \text{program} \rangle \) and distributed into \( \langle \text{classes} \rangle \) data structure. When this process
is finished, the execution begins with the invocation of method \text{main()} from
the class stored in \( \langle \text{mainClass} \rangle \). Cells \( \langle \text{program} \rangle \) and \( \langle \text{mainClass} \rangle \) represent
the initialization parameters of dynamic K-Java.

The reference documentation of each cell, along with the initial state of
the configuration, is given in the Appendix Section A.1.

### 6.2 Types and Values

Section 3.1 presented a programming language simple enough to be defined
using three ingredients: the syntax of IMP, the configuration and the notation
of \( K \). In order to define Java, we need one more ingredient: auxiliary notations
(or auxiliary syntax definitions), that allow us to express data structures and
intermediate states of computation not expressible otherwise. While K-Java
uses many auxiliary notations, in this section we focus on the most widely
Figure 6.2: Auxiliary syntax definitions for types and values
used ones: the notation for Java types and for Java values, encountered in
the majority of the rules from the following sections.

Types. The notation for types and values is displayed in Figure 6.2. Types
in K-Java are represented by the sort Type, that may be either a primitive
type (PrimitiveType), a reference type (RefType) or void. For the sake of
uniformity, we decided to consider void also a type, although technically it
is not, according to JLS. This allows us, for example, to have an uniform
representation of return types for all methods, by converting return statements
without arguments to return statements with argument of type void. A
PrimitiveType may be either boolean bool, or one of the seven options for
NumericType.

It is worth noting that there is some inconsistency between the syntax of
Java and some of the notations in this section that have an equivalent in Java,
such as bool being the representation of Java boolean. Part of this ambiguity
is caused by K-Java being defined in the AST form, part of it by the limitations
of K rule parser at the moment K-Java was under development. For parts
that have no equivalent in Java, such as the notation for ClassType, it shall
be considered auxiliary notation of K-Java. We suppose this inconsistency
will be addressed once K-Java will be rewritten in the syntactic form.

A reference type may be either a class type (ClassType), an array type
represented by arrayOf followed by the array element type, or a null type
nullType. The null type is the type corresponding to the value null; it is
defined in JLS but inaccessible directly to the programmers. A class type is
represented by the terminal class followed by the fully qualified name of the
class, e.g. the full package name followed by the class name. For example for
Object the K-Java class name would be class java.lang.Object, although
in K rules in this chapter we will use class Object for brevity.

Values. Java values are represented in K-Java by the sort RawVal. Primitive
values may be either decimal, floating-point or boolean, represented by the
sorts Int, Float and Bool respectively. All three sorts are part of K builtin
library; we do not have to define them in K-Java. Reference values have the
separate sort RawRefVal explained below. Also, there is the artificial value
nothing, the value corresponding to the type void.

Reference values, represented by RawRefVal may either be object refer-
ences, array references, instances of the $K$ builtin sort String or null references (null). An object reference is represented by the auxiliary notation $objectRef$ with two arguments — the object identifier in the memory and the runtime class of the object. The usage of this construct in rules will be demonstrated in Section 6.6. The representation of an array reference $arrayRef$ has three arguments: the type of its elements, the location of the first element in memory and the length.

String is defined in JLS as a simple class. Its only specific language-level feature is string literals, like "abc". However, since strings are ubiquitous in our test suite, we decided to model strings with the $K$ builtin sort $String$, for performance reasons. A few methods members of class $String$ that we support are implemented through $K$ rules rather than being Java code.

The null reference has the same meaning as in Java.

Typed values and computation result. Since Java is a statically typed language, every time we perform an operation over a value, we also need access to its static type, as was illustrated in Section 1.3. For this reason, a value in K-Java is never encountered alone as $RawVal$, but as part of a typed value construct $TypedVal$, consisting from raw value, followed by $::$ separator, followed by its static type. A $TypedVal$ in K-Java is the final computation result of an expression, for this reason it is subsorted to both expression sort $Exp$ and $KResult$. Recall that in $K$ computation results are represented by the sort $KResult$. Another major final value is $Type$. There are also other auxiliary notations that serve as $KResult$, we will introduce them by need.

More specialized auxiliary notations used in this chapter will be explained by need.

6.3 Expressions

We choose to illustrate expression semantics based on four operators: plus, type cast, conditional operator and $instanceof$ (Figure 6.3).

Strictness. Expressions are the part of the semantics that benefits the most from the strictness mechanism. Strictness annotations of the four expressions are displayed on the syntax definitions in the upper part of the figure.
SYNTAX $Exp ::= Exp + Exp$ [seqstrict]
| $(Type) Exp$ [strict]
| $Exp ? Exp : Exp$
| $Exp $instanceof $RefType$ [strict]

RULE PLUS-INT

$I1:Int :: — + I2:Int :: —$

$(I1 +_int I2) :: tempType$

RULE CAST-NUMBER

$(NT:NumericType) V:RawVal :: —$

$normalize (V :: NT)$

RULE CAST-REFTYPE

$(RT:RefType) V:RawRefVal :: —$

$if (subtype (typeof (V), RT) \{ V :: RT \})$
$else \{ throw new class ClassCastException (...) \}$

RULE CONDITIONAL


$if (CondE) \{ TrueE \} else \{ FalseE \}$

RULE INSTANCEOF

$V:RawVal :: — $instanceof $RT:RefType$

$(V \neq null) \&\& subtype (typeof (V), RT)$

Figure 6.3: Rules for selected expressions: $+_\_\_$, cast, $?_:\_$, $instanceof$
The order of evaluation of expression arguments in Java is always left to right: the for a binary operator, the left operand will be evaluated before the right one; for method invocations, arguments will be evaluated left o right. This behavior is typically achieved in a K-Java using seqstrict attribute, as is the case for plus operator. Recall that seqstrict means all operands have to be evaluated from left to right, each operand staying in the top of computation for evaluation until it reaches a value of sort KResult.

The correct specification of strictness may be easily tested by using expressions with side effects, where incorrect order of operand evaluation leads to incorrect order of side effects. This observation is used for testing the expressions.

Since for operators cast and instanceof just one operator may produce side effects (type evaluation does not produce side effects), both strict and seqstrict are equivalent. We choose to use strict. The conditional operator does not use the strictness mechanism.

Operator plus Once the operands of a _+_ expression are evaluated to TypedVal by the strictness mechanism, the rule for plus with integer operands (PLUS-INT) may apply. The plus expression is rewritten into a typed value with raw value being the sum of the two operands, by using the K builtin operator +Int. The static type of the result is the auxiliary notation tempType, which is a placeholder that will be explained shortly.

This rule needs a few explanations. K builtin operators, are "pure functions": their evaluation may happen in any context, they don’t need to be brought to the top of ⟨⟩k. For this reason the evaluation of +Int will happen "in place", without any additional specifications. Blue parentheses ( ) in the rule PLUS-INT represent the K notation for brackets, used for grouping and parsing disambiguation of K rules. In contrast, when black parentheses are encountered (such as in the cast expression), they represent part of some syntax defined in K-Java, either of Java or of auxiliary K-Java notations.

Such a semantics is sufficient, because, as a result of static preprocessing, each expression is wrapped into a cast expression (see Figure 5.2), and the execution of this cast expression will replace tempType with the expected type of the plus expression. Since the type of the result is already encoded into an enclosing cast, there is no use for the static types of operands in this case, they may be simply ignored.
**Type Cast.** The cast operator has three rules: one for numeric values, one for references and one for boolean expressions (which is trivial and not presented here).

The rule for numbers (**CAST-NUMBER**) rewrites the cast expression into the auxiliary function `normalize`. The argument of `normalize` is a typed value having as raw value the casted value `V`, and as type the cast target type `NT`. The function `normalize`, defined in Section B.4, implements integer overflow. It truncates `V`, if integer, to the value range covered by the type `NT`. In case `V` is a floating-point number, the normalization function has no effect. For floating-point numbers, we rely on their underlying implementation in K entirely, their precise representation with simple or double precision is not implemented in K-Java.

If the target type of the cast is reference, there are two cases. If the actual (dynamic) type of the raw value `V` is a subtype of the cast target type `RT`, the cast will evaluate into a typed value with raw value being the same `V` and static type being `RT`. No manipulation of `V` is necessary. If the dynamic type of `V` is not a subtype of `RT`, the cast is invalid, and a `ClassCastException` is thrown. Both cases are implemented in one single rule, by using a generalized definition of the Java `if` statement, that accepts any K terms as arguments 2 and 3. To extract the runtime type of a raw value we use the K-Java auxiliary function `typeOf`.

**Conditional operator.** The conditional expression is simply rewritten into a generalized `if`, as the two have the same dynamic semantics. The most tricky part of the conditional — determining the static type of the result, is already done by the static semantics. Any conditional expression reaching this stage will be wrapped into a cast encoding its static type, just like the plus expression.

**Instanceof operator.** The rule for `instanceof` relies on the already seen auxiliary function `subtype` to decide whether the raw value of the first argument is an instance of the type given as second argument. However, if target value is `null`, the operator always returns false, in conformance with JLS.

The rules presented in this and in the next section do not mention any cells. Recall that such rules are assumed to match at the beginning of $\langle \rangle_k$. 
SYNTAX \[ Stmt ::= \text{if} \ (Exp) \ Stmt \]
\[ \text{if} \ (Exp) \ Stmt \ \text{else} \ Stmt \ [\text{strict}(1)] \]

RULE IF-THEN-DESUGAR
\[
\text{if} \ (E : Exp) \ S : Stmt
\]
\[
\text{if} \ (E : Exp) \ S \ \text{else} \{/\}
\]

RULE IF-TRUE
\[
\text{if} \ (\text{true} : \text{bool}) \ S : \text{Stmt} \ \text{else} \ —
\]
\[
S
\]

RULE IF-FALSE
\[
\text{if} \ (\text{false} : \text{bool}) \ — \ \text{else} \ S : \text{Stmt}
\]
\[
S
\]

Figure 6.4: Rules and syntax for if statement

6.4 Statements

In this section we illustrate the semantics of statements based on the following list: if, blocks, while, break, continue, throw and try/catch. For each statement we discuss all the corner cases that required special design considerations, this includes a number of interesting interactions.

6.4.1 Statement if

If statement in Java comes in two versions: if-then and if-then-else (Figure 6.4). The version if-then is desugared into if-then-else with an empty block for else clause (rule IF-THEN-DESUGAR). It does not need any strictness annotation, as strictness is handled by if-then-else.

The if-then-else version is strict in its first argument. Once the argument is computed into a boolean typed value (e.g. \( V : \text{bool} \)), either the rule IF-TRUE or IF-FALSE applies, rewriting the if statement to the then clause, or else clause, respectively.
rule Block
\[
\begin{align*}
\{ \text{S:Stmt} \} \\
\text{S} \triangleright \text{env} (\text{Env}) \\
\end{align*}
\]

\[ \text{env} \quad \text{env} \quad \text{env} \quad \text{env} \]

rule Env
\[
\begin{align*}
\text{env} (\text{Env:Map}) \\
\text{env} \quad \text{env} \\
\end{align*}
\]

Figure 6.5: Rules for block statement

6.4.2 Statement Blocks

The semantics of statement blocks (Figure 6.5) must preserve the correct scope of local variables. According to JLS §6.3, the scope of a local variable declaration inside a block is the rest of the block in which the declaration appears. For this reason, the information about local variables must be saved at the beginning of a block, and restored at the end.

The rule Block is matched at the beginning of a block. It rewrites the block into its enclosing statement S followed by the auxiliary construct env, which saves a backup copy of the local variables environment env. We will show env in action later, in Sections 6.5 and 6.6. The configuration abstraction mechanism of K ensures that both env and env are matched in the same thread, in case there are multiple threads running.

Technically, the term matched by S in this rule is not a single statement, but a sequence of statements separated by ;. Hereafter we will consider such sequences as single statements for simplicity, both a statement and a sequence being of sort Stmt.

After the statement S execution is finished normally, the term env reaches the top of computation, and the rule Env is matched. This rule restores the environment to its state prior to block execution.

6.4.3 Statements While, Break, Continue

We illustrate the basic architecture for loops in K-Java on the example of while statement. The semantics of while, in addition to its main case, should also account for interaction with statements break and continue, also
RULE WHILE

\[
\text{while } (E:Exp) S:Stmt \\
\text{if } (E) \{ \text{S } \leadsto \text{ while } (E) \text{ S } \}
\]

RULE Break-Propagate

\[
\text{break } ; \leadsto \text{ KI:KItem} \leadsto 'K
\]

REQUIRES \¬_{\text{Bool}} \text{ interactsWithBreak } (\text{KI})

RULE AbruptTerminationStmt-Restore-Env

\[
S:Stmt \leadsto \text{ env } (\text{Env}:\text{Map}) \\
\text{env } (\text{Env}) \leadsto S
\]

REQUIRES isAbruptTermination (S)

RULE Break

\[
\text{break } ; \leadsto \text{ while } (\_\_\_) \_\_\_
\]

RULE Continue-Propagate

\[
\text{continue } ; \leadsto \text{ KI:KItem} \leadsto 'K
\]

REQUIRES \¬_{\text{Bool}} \text{ interactsWithContinue } (\text{KI})

RULE Continue

\[
\text{continue } ; \leadsto \text{ while } (\_\_\_) \_\_\_
\]

Figure 6.6: Rules for while, break and continue statements
presented here.

**While.** The first rule in Figure 6.6 implements the semantics of `while` through unrolling, similarly with the rule for `while` of IMP (Section 3.1). The while statement has no strictness arguments, as the expression `E` has to be evaluated again after each iteration.

**Break.** In Java, a statement or expression may complete normally or abruptly (JLS §14.1). Abrupt completion happens when the normal execution flow is interrupted, as a result of one of the statements `return`, `throw`, `break` or `continue`.

Statement `break` comes in two versions in Java: simple and labeled. Here we only discuss the simple `break`. This statement interrupts the innermost `while`, `for`, `do` or `switch` statement, passing the execution to the next statement after the interrupted one. The interaction of `break` with various statements requires several rules.

The rule `BREAK-PROPAGATE` represents the situation when the statement `break` with no label is in the top of computation, and is followed by a term with which it cannot interact. Ability to interact with `break` is implemented by the predicate `interactsWithBreak`, and is true only for statements `while`, `finally` clauses of `try`, `switch` and auxiliary construct `env`. To match only the next `K` term, we use the variable `KI` of sort `KItem`. If the variable would be of sort `K`, then the whole rest of computation would be matched, as the sort `K` generally represents a computation sequence of `KItem` terms separated by `↷`. When this rule matches, the term `KI` is dissolved, and `break` remains in the top of computation. The rule may be matched repeatedly, until the next term after `break` becomes a statement with which `break` can interact.

A special case of interaction with `break` is `env`. When `break` is followed by an `env`, this means `break` statement reached the end of a `{...}` block. In this case `env` construct should not be dissolved but executed, to ensure proper restoration of the local environment, a mandatory logic regardless if the block execution completed normally or abruptly.

The rule `ABRUTPTerminatinStmt-Restore-Env` implements such abrupt termination of a block, caused not only by `break` but by all statements causing abrupt termination, mentioned earlier. The predicated `isAbruptTermination` in the side condition is true for all four abrupt
termination statements. The rule just reorders the abrupt statement S and env term, letting env to reach the top of computation and execute. This is an example of subtle interaction of Java features, in this case the abrupt termination and block, carefully tested and exposed by our test suite.

The main rule for break (BREAK) matches when break is followed by a while statement. Then both break and while are dissolved, and execution continues with the next statement after while.

**Continue.** In Java, the statement continue (version without labels) interrupts the current iteration of the innermost loop and transfers the control to the beginning of the loop. The semantics for continue is given similarly to break. The rule CONTINUE-PROPAGATE dissolves the first statement after continue, as long as it does not interact with continue. Interacting statements are while, finally clauses of try, and env. The rule matches repeatedly, until a statement interacting with continue is reached.

The rule CONTINUE matches when continue is followed by a while. In contrast with break, this time only the continue statement is dissolved, the while statement is left unchanged, allowing it to execute from the next iteration.

The version of while presented here is simplified, being just enough to illustrate the interaction with unlabeled versions of break and continue, but having the advantage of being more intuitive. Java also has labeled versions of break and continue that allow jumping over several nested loop statements until a loop matching the label is found. The full semantics of while, break and continue is given in Appendix Section A.4.

### 6.4.4 Statements throw and try/catch

In this section we illustrate the semantics of statements responsible for exception handling: throw and try/catch (Figure 6.7). They exhibit an interaction similar to that of break and while.

The first rule rewrites a try/catch into the body of try, S, followed by catch clauses wrapped into the auxiliary construct catchBlocks. Now rules for other statements will apply over the content of S, until it is either entirely consumed or a throw statement becomes the first term.

In the first case, if S was consumed, it means the execution of try
RULE TRY-CATCH

\[
\begin{align*}
\text{try } & \text{S:Stmt } \text{CatchList:CatchClauses} \\
\hline
& \text{S } \leadsto \text{catchBlocks (CatchList)}
\end{align*}
\]

RULE CATCHBLOCKS-DISSOLVE

\[
\begin{align*}
\text{catchBlocks } (\rightarrow) \\
\hline
& \rightarrow
\end{align*}
\]

SYNTAX \[ Stmt ::= \text{throw Exp ; [ strict ]} \]
compiled into:

<table>
<thead>
<tr>
<th>RULE THROW-HEAT</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
& \rightarrow \\
& \text{E } \leadsto \text{throw E:Exp} \\
& \hline
& \text{E} \\
& \rightarrow \\
& \text{\text{catchBlocks} (\rightarrow)} \\
& \rightarrow
\end{align*}
| REQUIRES ~\text{Bool isTypedValue (E)} |

<table>
<thead>
<tr>
<th>RULE THROW-COOL</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
& \rightarrow \\
& \text{V:RawVal :: ThrowT:Type} \\
& \hline
& \text{\text{throw E:Exp}} \\
& \rightarrow \\
& \text{V} :: \text{ThrowT} \\
& \rightarrow \\
& \rightarrow
\end{align*}
| |

<table>
<thead>
<tr>
<th>RULE THROW-MATCH</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
& \rightarrow \\
& \text{throw V:RawVal :: ThrowT:Type} \\
& \hline
& \text{\text{catchBlocks} (\rightarrow)} \\
& \rightarrow \\
& \{ \text{CatchT X ; } \rightarrow \text{X = V :: CatchT ; } \rightarrow \text{S} \} \\
& \rightarrow \\
& \rightarrow
\end{align*}
| REQUIRES subtype (ThrowT, CatchT) |

<table>
<thead>
<tr>
<th>RULE THROW-NOT-MATCH</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
& \rightarrow \\
& \text{throw V:RawVal :: ThrowT:Type} \\
& \hline
& \text{\text{catchBlocks} (\rightarrow)} \\
& \rightarrow \\
& \{ \text{CatchT X ; } \rightarrow \text{X = V :: CatchT ; } \rightarrow \text{S} \} \\
& \rightarrow \\
& \rightarrow
\end{align*}
| REQUIRES ~\text{Bool subtype (ThrowT, CatchT)} |

<table>
<thead>
<tr>
<th>RULE THROW-PROPAGATION</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
& \rightarrow \\
& \text{throw KI:KItem :: \rightarrow Type} \\
& \hline
& \rightarrow \\
& \rightarrow \\
& \rightarrow
\end{align*}
| REQUIRES ~\text{Bool interactsWithThrow (KI)} |

Figure 6.7: Rules for try/catch and throw
block completed normally. Then the term `catchBlocks` reaches the top of computation and is discarded by the second rule. Now the entire `try/catch` is consumed and the execution continues with the next statement.

In case `S` throws an exception, a `throw` statement will eventually reach the top of the computation. Figure 6.7 continues with the syntax definition of `throw`, annotated with `strict`. The strictness attribute is compiled into one heating and one cooling rule, as illustrated in the inner box of the figure. There is nothing specific with this particular strictness, we just preferred to illustrate the strictness mechanism again.

After the argument of `throw` is evaluated into a typed value, the actual rules for `throw` are ready to apply. Rule 3 (`throw-match`) matches when the type of the exception carried by `throw` is compatible with the type accepted by the first `catch` clause, so the exception is caught. The compatibility of the exception type and the `catch` clause’s type is verified by the expression `subtype(ThrowT, CatchT)` in the side condition. In this case, both the `throw` statement and the `catchBlocks` construct are rewritten into a sequence of three statements: a declaration of the `catch` parameter `X` of type `CatchT` as a local variable (`CatchT X;`), an initialization of `X` with the exception value typed with the type expected by `X` (`X=V::CatchT;`), and the body of `catch` — `CatchS`. All three statements are wrapped inside a block `{ }`, to confine the variable `X` to its expected scope.

Rule 4 matches when the two types are not compatible. In this case the first `catch` clause is dissolved, bringing the next one (if any) to the top of the list. The notation `·` here represents an empty element of the syntactic list `CatchClauses`. Rewriting an element of this list to `·` means deleting the element. This is in fact the same notation as `·K`, the empty element of a list of sort `K` of items separated by `↷`.

Rule 5 matches when `throw` cannot interact with the next statement (the side condition). Statements interacting with `throw` are the non-empty `catchBlocks` and `finally`. Thus, if the next computation item after `throw`, `KI:KItem`, is anything except the two, including the empty `catchBlocks` produced eventually by the fourth rule, that statement will be dissolved. The statement `throw` will remain the top of computation until it reaches a matching `catch` or remains the only rule inside the `{ }k` cell. The sort of `KI:KItem` ensures that variable `KI` matches only one item in the computation sequence separated by `↷`. If `throw` remains the only task in the computation,
another rule will match that will ensure exception propagation outside the current method.

One may wonder why not desugaring a try with multiple catch clauses into multiple try with one catch each. We actually did so initially, but it turned out to be incorrect. Indeed, the code

```java
try { throw new A(); }
catch (A a) { throw new B(); }
catch (B b) { ... }
```

cannot be desugared into

```java
try {
    try { throw new A(); }
    catch (A a) { throw new B(); }
}
catch (B b) { ... }
```

because in the second case the exception B would get caught by catch(B b), and this is incorrect according to JLS. This (counter) example is now included in our test suite.

6.5 Memory Model

Our memory is essentially a map from symbolic memory locations to typed values, held in the cell ⟨⟨⟨⟩⟩⟩store. Since Java does not allow pointer arithmetic, locations may be regarded as symbolic numbers. Also, because there is no capability of byte-level access to memory, each location stores a whole value. There is no need to divide values into bytes, as was done, for example, in C semantics [18].

Besides ⟨⟨⟨⟩⟩⟩store, there are many other cells related to memory, used to map various sorts of variables to their memory locations and to store the information required for static initialization and object instantiation. In the remaining of this section we describe all the cells involved in accessing the memory, while the actual rules using these cells will be given in the next two sections.

For better clarity, the fragment of the configuration directly related to memory model was extracted to Figure 6.8. The cell ⟨⟨⟨⟩⟩⟩methodContext inside ⟨⟨⟨⟩⟩⟩thread groups the information specific to the currently invoked method.
from all the cells contained in \( \langle \rangle \text{class} \), here we only mention those involved in the memory model. The fully qualified class name is held in \( \langle \rangle \text{classType} \), acting as identifier. Two cells important in object instantiation are \( \langle \rangle \text{enclosingClass} \) and \( \langle \rangle \text{extends} \). The cell \( \langle \rangle \text{enclosingClass} \) holds the directly enclosing class of the current class, in case the current class is an inner class, remaining empty otherwise. The cell \( \langle \rangle \text{extends} \) stores the base class. The two represent directly dependent classes of the current class. The cell \( \langle \rangle \text{instanceFields} \) contains field declarations for all instance (e.g. non-static) fields of this class, in the form \( T1 \, V1; \, T2 \, V2; \) etc; this content is simply executed during object instantiation. The last cell inside \( \langle \rangle \text{class} \) important for variable access is \( \langle \rangle \text{staticEnv} \) — a map from static fields to their locations in \( \langle \rangle \text{store} \), populated during static initialization of the class.

The next two cells in the figure are the already mentioned \( \langle \rangle \text{store} \), and the locations counter \( \langle \rangle \text{nextLoc} \), used by variable declarations.
The remaining big group of cells — \( \langle \text{objectStore} \rangle \) contains the inner structure of the objects. It contains one \( \langle \text{object} \rangle \) for each object instantiated by the program. Here \( \langle \text{objectId} \rangle \) is an unique identifier of the object, also used as store location. Inside \( \langle \text{objectType} \rangle \) the actual runtime type of the object is held. The same information is held by every object reference of the form \( \text{objectRef}(\text{OId}, \text{C}) \), with \( \text{Oid} \) being the object id and \( \text{C} \) being the runtime type being. This duplication is probably not necessary, yet such suboptimal choices are the consequence of the large scale of the present work.

To facilitate the semantics of fields, objects have a layered structure, each layer representing one class in the inheritance hierarchy. For example an object of type \( \text{B} \), with \( \text{B} \) derived from \( \text{A} \) derived from \( \text{Object} \), will have three layers: one for each of the \( \text{Object} \), \( \text{A} \), \( \text{B} \). Java interfaces cannot have instance fields, for this reason they are not represented in the layers. Each layer, represented by \( \langle \text{layer} \rangle \), has three subcells: the corresponding class \( \langle \text{layerClass} \rangle \), the instance fields environment \( \langle \text{layerEnv} \rangle \) and the reference to the enclosing object \( \langle \text{layerEnclosingObject} \rangle \) (for non-static inner classes). Java has complex rules related to inner classes (JLS §8.1.2). For non-static inner classes, an object has special access to an enclosing object an instance of the enclosing class. If multiple layers are non-static inner classes, then each of them has a separate, and potentially different enclosing object. Inner classes, however, are not detailed in this thesis.

An example configuration fragment for an object with the aforementioned hierarchy (\( \text{Object} > \text{A} > \text{B} \)) is given in Figure 6.9 along with the source code for class declarations. The object store contains one instance of \( \text{B} \), illustrating the layers. The \( \langle \text{store} \rangle \) contains three entries — one for the object and two for its fields. The figure also illustrates a field access expression in the middle of evaluation, with the qualifier already evaluated, inside \( \langle \text{k} \rangle \).

### 6.6 Variables

In this section we give the rules for the majority of the components of variable access logic. Java has three types of variables based on the scope of their definition: local variables, instance fields and static fields. As a result of preprocessing, all three categories may be distinguished syntactically. Only local variables are referred by simple name in dynamic K-Java. Instance
class A { int v1;}
class B extends A { int v2;}

Figure 6.9: An example fragment of configuration containing inheritance and an object instance.
fields and static fields are qualified with a reference to an object or a class respectively, as shown in Figure 5.2 row 4.

In this section we cover local variable declaration, lookup of all three categories of variables, and the assignment operator for local variables. Starting with this section we show more advanced rules, that heavily rely on configuration cells.

### 6.6.1 Local Variable Declaration

In dynamic K-Java variable declarations always come in their simplest form:

```
Type Var;
```

Although Java allows defining more complex forms, such as multiple variables within one declaration, declarations with initializers and even C-style array declarations (JLS §14.4), all of them are desugared into simple variable declarations and assignments by the static K-Java.

The local variable declaration is implemented with one rule, given in Figure 6.10. A new entry is allocated in \( \{ \text{store} \} \), with symbolic location being the current value of \( \{ \text{nextLoc} \} \), and value being the default value for the type \( T \). The default values for types are defined in the JLS, those are 0 for numeric types, `false` for boolean and `null` for references. The value of \( \{ \text{nextLoc} \} \) is incremented to point to a fresh value for a future allocation. The local environment \( \{ \text{env} \} \) is updated with a new entry associating the newly declared variable \( X \) to its store location \( L \). In the computation, the declaration is dissolved.

It is easy to notice that without a deallocation mechanism the size of \( \{ \text{store} \} \) grows with every variable declaration, which may cause performance
Figure 6.11: Rules for local variable lookup

problems for large programs. Our semantics does not implement any store cleanup mechanisms, as performance was not the focus. Such mechanisms may be easily added at some future point. The same goes for the object store — objects are never deallocated in K-Java. That is, we do not have a garbage collector. This is indeed a valid behavior; according to JLS garbage collector is optional.

6.6.2 Local Variable Lookup

The local variable lookup is implemented by two rules (Figure 6.11). The first rule looks up the store location $L$ for the variable $X$ and rewrites the identifier into the auxiliary construct $\text{typedLookup}(L)$. The second rule implements $\text{typedLookup}$ — the construct is rewritten into the typed value referred by the store location $L$.

These two rules could be easily replaced by one. Yet this separation allows us to reuse $\text{typedLookup}$ logic, common for all kinds of variable lookup.

6.6.3 Field Lookup

The semantics for field lookup is given in Figure 6.12.

Instance Field Lookup. The instance field access expression is strict in its first attribute, triggering the evaluation of the field qualifier. Once the qualifier is evaluated to a non-null value, it will have the form $\text{objectRef}(\ldots)::C$. At this point the rule $\text{FIELD-INSTANCE}$ applies, rewriting the object reference
SYNTAX  \[\begin{align*}
\text{Exp} & ::= \text{Exp} \cdot \text{Id} \ [\text{strict}(1)] \\
& | \ \text{ClassType} \cdot \text{Id}
\end{align*}\]

**RULE Field-instance**

\[
\begin{array}{c}
\langle \text{objectRef (OId:Int, ―) :: C:ClassType} \cdot \text{X:Id} \rangle \\
\text{typedLookup (L)}
\end{array}
\]

\[
\langle \text{OId \_objectId \ C \_layerClass \ X \_L:Int \_layerEnv} \rangle
\]

**RULE Field-instance-OfNull**

\[
\langle \text{null :: ―} \rangle \\
\text{throw new class NullPointerException (...)}
\]

**RULE Field-static**

\[
\langle C:ClassType \cdot \text{X:Id} \rangle \\
\text{staticInit (C)} \bowtie \text{staticFieldLookup (C, X)}
\]

**RULE staticFieldLookup**

\[
\begin{array}{c}
\langle \text{staticFieldLookup (C:ClassType, X:Id)} \rangle \\
\text{typedLookup (L)}
\end{array}
\]

\[
\langle \text{... X:Id \_L:Int \_staticEnv} \rangle
\]

Figure 6.12: Rules for field lookup
into the already seen typedLookup construct. This time, in order to find the correct location $L$, a different set of cells is employed. We have to find the object in the object store, and for that object, to match the layer corresponding to the static type of the object reference.

This is perhaps the best example of configuration abstraction so far. The rule specifies only the leaf cells, leaving the configuration abstraction to infer the context. The resulting pattern, for the three cells related to object store, is the following:

\[
\begin{align*}
\langle \text{OID} \rangle_{\text{objectId}} & \quad \langle C \rangle_{\text{layerClass}} \\
\ldots \quad X \mapsto L: \text{Int} & \quad \langle \text{layerEnv} \rangle_{\text{layerEnv}} \\
\langle \text{objectStore} \rangle_{\text{objectStore}} & 
\end{align*}
\]

It is also possible that the qualifier will evaluate to null. In this case the rule Field-instance-OfNull will match, throwing a null pointer exception.

**Static Field Lookup.** There are two rules specific to static fields, applied sequentially. The first rule (FIELD-STATIC) rewrites the field access expression into the wrapper staticFieldLookup that has the same information as the field access, preceded by the staticInit function. The purpose of this rule is to trigger the static initialization of the class, in case it was not initialized already.

According to JLS, static initialization happens when one of the following events for a class happens for the first time: a static field is accessed, a static method is called or the class is instantiated. Its role is to allocate static fields and to execute static initializers. There are many other rules related to static initialization, all of them are implemented by staticInit, and extracted to Appendix Section A.12.

The second rule (STATICFIELDLOOKUP) performs the actual field lookup, acquiring the field location from the static environment $\langle \text{staticEnv} \rangle$ of the qualifier class. We could not inline the logic of the second rule into the first one, because the static environment is populated during static initialization.

### 6.6.4 Assignment Operator

The semantics of assignment may be divided into three steps: evaluation of the left-hand side (LHS), evaluation of the right-hand side (RHS), and the
RULE ASSIGN-HEAT
\[
\frac{\kappa \cdot \text{lvalue}(E)}{\text{lvalue}(E) \rightsquigarrow E : \text{Exp} = \square}
\]
REQUIRES $\neg_{\text{Bool}} \text{isTypedVal}(E)$

RULE LVALUE-LOCAL-VAR
\[
\frac{\text{lvalue}(\mathbf{X} : \mathbf{Id}) \quad \text{loc}(\mathbf{L} : \mathbf{Int}) :: \mathbf{T} : \mathbf{Type}}{
\kappa \langle \mathbf{X} \mapsto \mathbf{L} \rangle_{\text{env}} \langle \mathbf{L} \mapsto \square :: \mathbf{T} \rangle_{\text{store}}
}
\]

SYNTAX \[\text{RawVal ::= loc(\mathbf{Int})}\]

RULE ASSIGN-COOL
\[
\frac{\mathbf{T} : \text{TypedVal}}{\kappa \cdot \mathbf{T} \cdot \mathbf{loc}(\mathbf{L}) :: \mathbf{T}}
\]

CONTEXT \[\mathbf{T} : \text{TypedVal} = \square\]
compiled into:

\[
\frac{\kappa \cdot \mathbf{E} \cdot \mathbf{loc}(\mathbf{L}) :: \mathbf{T}}{\text{lvalue}(\mathbf{E}) \rightsquigarrow \mathbf{E} : \text{Exp} = \square}
\]
REQUIRES $\neg_{\text{Bool}} \text{isTypedVal}(\mathbf{E})$

RULE
\[
\frac{\kappa \cdot \mathbf{T} \cdot \mathbf{loc}(\mathbf{L}) :: \mathbf{T}}{\text{lvalue}(\mathbf{T}) \rightsquigarrow \mathbf{T} : \text{TypedVal} = \square}
\]

RULE ASSIGN
\[
\frac{\text{loc}(\mathbf{L} : \mathbf{Int}) :: \square = \mathbf{T} : \text{TypedVal}}{\mathbf{T} : \text{TypedVal} \rightsquigarrow \mathbf{L} \mapsto \square :: \mathbf{T}}
\]
\[
\kappa \langle \mathbf{L} \mapsto \square :: \mathbf{T} \rangle_{\text{store}}
\]

Figure 6.13: Rules for assignment
actual assignment operation.

**Evaluation of the LHS.** This part cannot be defined by strictness, because LHS has to be evaluated up to a store location only. Simply heating LHS by strictness would result in its evaluation all the way up to a typed value. Instead, we use custom heating/cooling rules with wrappers, that allow us to control the level to which LHS expression is evaluated (Figure 6.13).

The first rule (ASSIGN-HEAT) heats the LHS wrapped into the auxiliary notation lvalue. If the LHS is a reference to a local variable, the rule LVALUE-LOCAL-VAR would apply, evaluating the term up to a 'special typed value': loc(L) :: T. Here loc is another auxiliary notation, whose primary role is to prevent the evaluation of the term any further. Similar rules exist for other forms of LHS, such as fields or array access expressions.

Now the rule ASSIGN-COOL could apply, cooling the special typed value back to its place in the assignment operator. The first computation item matches a TypedVal because loc is subsorted in our definition to RawVal. This heating/cooling mechanism is inspired from the language SIMPLE [44].

**Evaluation of the RHS.** For the RHS we also cannot use plain strictness, because RHS has to be evaluated only after the LHS, and there is no way to specify this with strictness attributes. Instead, K offers a generalization of the strictness mechanism, called context rules.

Context rules are special K rules that may contain rule variables and side conditions, but cannot refer to cells and cannot have rewrites. Also, one of their variables has to be the special variable □ (spelled hole) that indicates the place where heating/cooling should happen. A context rule is compiled into one heating and one cooling rule, that heat/cool the position marked by the hole.

For our assignment logic, the context rule for RHS evaluation is the fourth rule in Figure 6.13, the one starting with K keyword CONTEXT. The heating/cooling rules into which it is compiled are given in the inner box.

**Actual assignment.** The last rule in the figure is for actual assignment, matched when both LHS and RHS have been evaluated to TypedVal. Here the LHS store location is updated, and the assignment expression is rewritten to RHS in the computation. The RHS is still needed in ⟨⟩k, because in
Java assignment is an expression, not a statement, and may, for example, be chained like `a=b=c`.

As can be seen, only `lvalue-local-var` is specific to a particular type of LHS expression, the rest of the assignment rules are common. Although JLS allows the types of LHS and RHS to differ, requiring additional conversions, due to our static semantics such conversions are already encoded into cast expressions, thus no extra rules are needed.

The full rules for lookup and assignment are a bit more complex and fragmented into more steps, particularly due to arrays. In this section we presented a "cleaned up" version of them, just enough to illustrate the core semantics of local variables and fields. The full semantics is split across multiple sections of the Appendix, the main part being located in Sections A.6 and A.7.

### 6.7 New Instance Operator

New instance creation expressions in Java come in two forms: unqualified and qualified. Their definition in K-Java is given in the upper part of Figure [6.14]. They contain a slight difference comparing to Java: for the qualified version, Java allows only a simple class name (`Id`) as a second non-terminal, while K-Java also allows a `ClassType`. This distinction is required by K-Java rules.

The presence of two forms is a consequence of Java supporting instance inner classes. Unqualified `new` is used to instantiate top-level classes, static inner classes and, in case enclosing object may be inferred from the instantiation context, instance inner classes. Qualified `new` is only used for instance inner classes, in this case the qualifier explicitly specifies the enclosing object.

Static K-Java performs a partial normalization, in an attempt to reduce the number of cases. All `new` expressions for instance inner classes are converted into the qualified form. For other class types, `new` expressions are left unqualified.

Our deliberate restriction to represent the preprocessed AST as valid Java enforces us not to convert all the `new` expressions to the qualified form statically. This choice also brings a benefit: whenever K-Java needs to throw an exception, the exception can be instantiated in the natural unqualified form, such as in the rule `FIELD-INSTANCE-OFNULL` in Figure [6.12].
SYNTAX \( E_{xp} ::= \text{new } \text{ClassType} \left( E_{xps} \right) \)
\[\ | \ E_{xp} \cdot \text{new } \text{ClassOrName} \left( E_{xps} \right) \] [seqstrict(1,3)]

SYNTAX \( \text{ClassOrName} ::= \text{Id} \)
\[\ | \ \text{ClassType} \]

SYNTAX \( E_{xps} ::= \text{List}\{ E_{xp} \text{, } , \text{"} \} \) [seqstrict]

SYNTAX \( \text{TypedVals} ::= \text{List}\{ \text{TypedVal} \text{, } , \text{"} \} \)

SYNTAX \( E_{xps} ::= \text{TypedVals} \)

SYNTAX \( K_{Result} ::= \text{TypedVals} \)

RULE \text{NewInstance-to-QNewInstance-unpack}

\[
\text{new } C:\text{ClassType}(\text{ArgExps}:E_{xps}) \\
\text{nothing}::\text{void} . \text{new } C(\text{ArgExps})
\]

RULE \text{Qualified-new-instance-resolve-class}

\[
\text{Qual} . \text{new } \text{Name}:\text{Id} \\
\text{getClassType} ( \text{toPackage} ( \text{typeof} (\text{Qual})) , \text{Name})
\]

Figure 6.14: Normalization of new instance operator and evaluation of the arguments
**Definition of `new` in JLS.** The evaluation steps for `new` defined in JLS are the following. First the qualifier is evaluated. Then, the arguments are evaluated, not only for the current constructor but for the whole chain of constructor calls, starting with the invoked constructor down to the constructor for `Object`, according to the inheritance chain. Since Java allows one constructor to call another constructor of the same class through `this(...)`, for some classes in the hierarchy the chain may contain more than one constructor.

If the evaluation so far didn’t throw any exception, the actual object allocation happens, layer by layer, starting from the root class `Object` up to the instantiated class. For each layer, first the fields are allocated, then all instance initializers are executed, and finally constructors are executed, with their arguments being already evaluated.

**Normalization in dynamic K-Java.** The first step in the evaluation of `new` in the dynamic K-Java is the complete normalization to a single form. It is performed by the two rules in Figure 6.14. The first rule matches the unqualified `new` and adds a placeholder qualifier `nothing::void`. The second rule matches the qualified `new` and evaluates the simple class name to a fully qualified name, through a sequence of auxiliary functions. As a result, the form of `new` after normalization is always qualified, with the class to be instantiated converted to a proper `ClassType`.

**Strictness.** After normalization, the strictness rules apply (see syntax definitions in the figure), and evaluate first the qualifier, then the constructor arguments. The last four syntax definitions in the figure are also involved in the process. They define the syntactic list of expressions `Exps` and the list of typed values `TypedVals`. These definitions represent a list of arguments to a constructor or a method, as either initial expressions (`Exps`) or final values (`TypedVals`). The definition of `Exps` is annotated with `seqstrict`, thus ensuring evaluation of the arguments left-to-right, as specified in JLS.

**Main rules.** After the components of the `new` expression have been evaluated, the main logic of `new` is executed (Figure 6.15). The first rule to match is `QUALIFIED-NEW-INSTANCE`; it performs initial memory allocation and schedules the remaining steps. At this point the initial object structure is allocated in `⟨⟨⟨⟩⟩⟩store` and `⟨⟨⟨⟩⟩⟩objectStore`. The `⟨⟩methodContext` content is
Figure 6.15: Main rules for new instance operator
replaced with a fresh context required by the following steps.

In \( \{\} \), a series of steps are scheduled as a sequence of computation items separated by \( \leadsto \). Those are the potential static initialization of the class (staticInit()), the allocation of the layers (create()), potential setting up of the enclosing object for the instantiated class (setEncloser()), and invocation of the constructor (explained later). The remaining two computation items ensure that new evaluation ends in a proper state. Those are restoration of the original method context restoreMethContext() and and a typedLookup() evaluating to a reference to the produced object, the result value of new.

The remaining of Figure 6.15 represents the step create(), the allocation of object layers and consequently of fields. The rule CREATE defines the construct as a recursive function. Term create(C) is rewritten into a sequence of four terms: create() for the base class of C, setting up the current class as C, evaluation of field declarations FieldDecs and allocation of a new object layer. The recursion ensures layers are actually allocated bottom-up: starting from object up to the instantiated class.

The second rule for create (CREATE-EMPTY-DISCARD) is the last step in the recursion, called by the create(Object). Here we use \( \kappa \) as abuse of notation for the missing base class of Object. The rule for setCrntClass simply sets the current class, to be used later by addEnvLayer.

The field declarations FieldDecs mentioned in CREATE are all in their simplest form without initializers: Type X; same as local variable declarations in Section 6.6.1. They are executed by the rule for local variable declarations, consequently at the end fields map is stored in \( \{\} \). Since the declarations in FieldDecs have no initializers, no side effects may happen at this stage.

The last rule, ADDENVLAYER, allocates a new \( \{\} \) layer cell in the \( \{\} \) object representing the processed object. The \( \{\} \) layerEnv for this layer is initialized with the content of \( \{\} \), just populated earlier. The K-Java architecture for object allocation is inspired from the language KOOL [32].

**Constructor Invocation.** In the rule QUALIFIED-NEW-INSTANCE, the 4th item in the RHS of the first rewrite represents the constructor invocation. This is actually a method call expression, with some of the components being K-Java auxiliary constructs. In particular, getConsName(C) evaluates into the K-Java internal method name used for constructors of C.
Constructors are represented in the configuration as extended methods, carefully encoded to precisely implement the correct order of side effects according to the JLS. The only extension, comparing to plain methods, is the invocations of the next constructor in the chain, of the form `this(...);` or `super(...);`. This invocation is present on the first line of each constructor, except for `Object`. The instance initializers, that also have to execute at this stage, don’t need any special semantics in dynamic K-Java, as they are statically preprocessed into plain methods (Figure 5.2, row 6).

As a result of our constructor representation, even though the order of computation steps in K-Java is different than the one in JLS (described at the beginning of this section), the order of observable side effects is carefully preserved and tested, ensuring that our definition is JLS-compliant.

The full semantics for `new`, including the semantics for `this();` and `super();`, is given in Appendix Sections A.8 and A.9.

### 6.8 Method Invocation

Method invocation is arguably the most complex component of both static and dynamic K-Java. In static K-Java, method invocations are preprocessed to infer the expected argument types, in the presence of overloading. Moreover, overloading involves a complex interaction with access modes, as illustrated in Section 1.3. As a result, in dynamic K-Java the method signature is encoded in the type of arguments, and method calls are always qualified.

Yet there are many particular cases remaining, depending on whether qualifier object static type is a class or an interface, whether the method scope is instance or static, and again depending on access modes. Access modes, and particularly `package` access mode, play a complex role in conjunction with polymorphism.

In this section we give an almost complete semantics for dynamic method invocation. We start with some preliminaries describing new configuration cells. Following are four subsections discussing the four steps of method invocation logic:

- Evaluation of subexpressions, e.g. the qualifier and the arguments.
• Loading method information, an auxiliary step which loads static information about the method into the computation.

• Lookup method reference. At this stage polymorphism logic is defined. The result of this stage is a method reference, for which both the signature and the method definition (e.g. method body) are known.

• Actual method invocation.

The full semantics is given in Appendix Sections A.10 and A.11.

6.8.1 Preliminaries

Here we enumerate the new configuration cells which are used by method invocation rules (see Figure 6.1 for the configuration):

• \( \langle \rangle_{\text{stack}} \) inside \( \langle \rangle_{\text{thread}} \) — the method call stack. The cell content is a list of pairs of the form \( (K, Bag) \), with the first element being the content of \( \langle \rangle_{k} \), the second - the content of \( \langle \rangle_{\text{methodContext}} \), at the moment the stack entry was created.

• \( \langle \rangle_{\text{classMetaType}} \) inside \( \langle \rangle_{\text{class}} \) — May be either class or interface, a token of the sort ClassMetaType.

• \( \langle \rangle_{\text{methods}} \) inside \( \langle \rangle_{\text{class}} \) — The map of accessible methods in the class. Map keys are method signatures, map values are classes where the respective signatures are declared. This map contains both the methods defined in the current class as well as methods inherited from the base classes (but not from base interfaces).

• The collection of cells \( \langle \rangle_{\text{methodDec}} \) inside \( \langle \rangle_{\text{md}} \) inside \( \langle \rangle_{\text{class}} \) — represents only the methods declared in the current class. The subcells hold the entire method content:
  - \( \langle \rangle_{\text{methodSignature}} \) — The method signature, acting as identifier.
  - \( \langle \rangle_{\text{methodParams}} \) — The list of method parameters.
  - \( \langle \rangle_{\text{methodBody}} \) — The method body.
  - \( \langle \rangle_{\text{methodAccessMode}} \) — The method access mode, such as private or public.
- \langle methodContextType \rangle — The method context type, one of instance or static.

A method is identified in the above cells by its signature, defined as follows:

**SYNTAX**  
Signature ::= \texttt{sig} (Id, Types)

Here \textit{Id} is the method name, and \textit{Types} is a comma-separated list of parameter types.

### 6.8.2 Evaluation of the Subexpressions

According to JLS, the first step in method invocation is evaluation of the qualifier (§15.12.4.1), followed by the evaluation of the arguments (§15.12.4.2), left-to-right. Even if the qualifier evaluates to \texttt{null}, the arguments still have to be evaluated.

In K-Java, this is simply achieved by strictness annotations, similar to the annotations for \texttt{new}:

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

**SYNTAX** 
MethodName ::= Id

The method qualifier in the syntax above is of sort \textit{K}, as it might be either an expression or a class (for static methods). The method name has the intermediary sort \textit{MethodName}, to allow the rules in this section to replace the \textit{Id} with some auxiliary constructs defined in the following sections.

### 6.8.3 Loading Method Information

A method invocation expression in dynamic K-Java may have one of the following forms, distinguishable statically:

- A static method qualified by a class: \texttt{Class.f(args)}. The class in this case is ensured by static K-Java to be precisely the class defining the method, although JLS also allows a derived one.

- A static method qualified by an expression: \texttt{o.f(args)}. Even though the method is static, it may be qualified by an expression instead of a class. We cannot simply replace the expression by its static type, for this expression requires evaluation, possibly producing side effects.
• An instance method qualified by an expression, with expression static
type being a class: o.f(args)

• An instance method qualified by an expression, with expression static
type being an interface: o.f(args)

The purpose of this step is to load into the computation all the static
information that might be non-uniformly represented in the configuration for
the four categories above. This allows us to simplify the step that follows in
two ways: by reducing the cases above to a smaller number, and by reducing
the number of references to configuration cells. The current step is partially
required as a result of our restriction to have preprocessed AST valid Java
only. Without this restriction, it would probably be possible to represent
method calls more uniformly right from the static K-Java.

Rules. During loading method information step the method name in the
invocation expression is replaced by the auxiliary construct methodInfo(),
defined in Figure 6.16.

The methodInfo has 6 arguments:

• Method signature

• Qualifier static class

• Qualifier class meta type (class or interface)

• Method declaration class

• Method declaration context type (static or instance)

• Method declaration access modifier (private, package, protected or
public). For the purpose of uniformity we use the K-Java notation
package when no access modifier is provided.

From here on we make a distinction between the following two wordings:

Definition. Method declaration class is the type from which the method
declaration is visible to the static type of the qualifier. It may either be the
qualifier static type, if it declares the invoked method, or a base class/interface
of the qualifier static type.
**Definition.** *Method definition class* is the class containing the actual method body to be invoked. *In the absence of polymorphism the two classes are always the same. When polymorphism is involved, definition class may be a type derived from the declaration class.*

The first rule in the figure rewrites the method name into a `MethodInfo` and computes its first two arguments, by relying on some auxiliary functions. Matching the arguments as `Args:TypedVals` ensures that strictness rules already applied, and arguments are evaluated. For the sake of simplicity, we will consider the remaining four arguments optional and will use `·K` to denote lack of a value. They are computed by the following rules.

Next, there are two cases: the method is declared in either a class or an interface. The second rule matches when the method is declared inside a class. In this case, there is an entry for the signature `Sig` in the `Methods` cell of the qualifier class. We match that entry to extract the declaration class `DecC`. The third rule follows immediately, computing context type and access mode based on `DecC`. The two rules cannot be combined into one, as `DecC` may be the same as `QualC`, but might also be different. The two situations cannot be expressed by one match pattern.

The last rule matches right after the first, if the declaring class for this method call is in an interface. This might happen either if qualifier static type is an interface, or if qualifier is an abstract class extending an interface that declares this method. In both cases, `Methods` does not inherit declarations from base interfaces (this is a consequence of the unfolding phase). As a result, there will be no entry for `Sig` there, as checked by the rule side condition. Since JLS only allows instance and public methods inside interfaces, here the last two arguments of the `MethodInfo` may be inferred without matching them in the configuration.

### 6.8.4 Lookup Method Reference

During this step the actual class containing the method body to be invoked is chosen. We refer to this class as Method definition class, denoted `DefC`. It is here that polymorphism (or method overriding) rules of Java are implemented.

**JLS Definition.** The definition of overriding in JLS is rather complex and scattered across several chapters. The most relevant part is quoted below.
SYNTAX \[ \text{MethodName} ::= \text{MethodInfo}( \text{Signature}, \text{ClassType}, \text{ClassMetaType}, \text{ClassType}, \text{ContextType}, \text{AccessMode}) \]

rule Invoke-compute-methodInfo-Signature

\[ \text{Qual} : \text{KResult} \]

\[ \text{Name} : \text{Id} \]

\[ \text{methodInfo} \left( \text{sig (Name, getTypes (Args)), typeof (Qual)}, \right) \]

\[ \text{Args} : \text{TypedVals} \]

rule Invoke-compute-methodInfo-DecC

\[ \text{Sig} : \text{Signature}, \text{QualC} : \text{ClassType}, \]

\[ \text{MetaT}, \text{DecC} \]

\[ \text{QualC} : \text{ClassType} \]

\[ \text{MetaT} : \text{ClassMetaType} \]

\[ \text{DecC} : \text{ClassType} \]

\[ \text{ContextT} \]

\[ \text{Acc} : \text{AccessMode} \]

\[ \text{methodSignature} \]

\[ \text{methodContextType} \]

\[ \text{methodAccessMode} \]

rule Invoke-compute-methodInfo-ContextType

\[ \text{Sig} : \text{Signature}, \text{DecC} : \text{ClassType}, \]

\[ \text{ContextT}, \text{Acc} \]

\[ \text{DecC} : \text{ClassType} \]

\[ \text{Sig} \]

\[ \text{methodSignature} \]

\[ \text{methodContextType} \]

\[ \text{methodAccessMode} \]

rule Invoke-compute-methodInfo-ContextType-unmapped

\[ \text{Sig} : \text{Signature}, \text{QualC} : \text{ClassType}, \]

\[ \text{MetaT}, \text{instance}, \text{public} \]

\[ \text{QualC} : \text{ClassType} \]

\[ \text{MetaT} : \text{ClassMetaType} \]

\[ \text{Methods} : \text{Map} \]

REQUIRES \[ \neg \text{Bool} \text{ Sig in keys ( Methods) \]

Figure 6.16: Loading method information - syntax and rules
A class *inherits* from its direct superclass and direct superinterfaces all the non-private methods (whether abstract or not) of the superclass and superinterfaces that are *accessible* to code in the class and are neither overridden [for instance methods] nor hidden [for static methods] by a declaration in the class.

An instance method *m1* declared in a class *C* *overrides* another method with the same signature, *m2*, declared in class *A* iff both:

1. *C* is a subclass of *A*.
2. Either
   - *m2* is non-private and *accessible* from *C*, or
   - *m1* overrides a method *m3*, *m3* distinct from *m1*, *m3* distinct from *m2*, such that *m3* overrides *m2*.

The rules for inheritance imply that for any given signature, only the last method in the inheritance chain, accessible to a class *C*, is inherited by the class. We will refer to this method as "visible" to *C*.

Let us consider a non-private instance method call expression with signature *Sig*, static class of the target *SC* and dynamic class *DC*. Then, the method definition class *DefC* for this expression will be the most derived class in the inheritance chain of *DC* that defines a method with *Sig*, possibly overriding the version visible to *SC* (JLS §15.12.4.4).

One might (rightfully) ask why the specification is so complex. The answer is the reference to *accessibility* (e.g. access modifiers) in the definition of both inheritance and overriding. In fact, rules deduced from the specification above are quite simple for all the cases except one:

- Both *static* and *private* methods do not participate in overriding. For them, *DefC* is the same as *DecC* computed in the previous subsection.
- For *protected* or *public* methods, *DefC* is the most derived class in the chain of *DC* that contains a method with the right signature.
- The most complex case is for methods with *package* access mode, as here accessibility also depends on the package in which each class in the *DC* chain is defined.
package a;
public class A {
    void f() { ... }
    void g() { ... }
}

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

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

((A) new C()).f(); // executes B.f()
((A) new C()).g(); // executes C.g()

Figure 6.17: Overriding example with package access mode
This complex case is best illustrated on an example (Figure 6.17).

For the first call, \(((A)\text{ new } C()).f()\), neither \(A.f()\), nor \(B.f()\) are accessible from the class \(C\). For this reason, \(C.f()\) cannot override \(A.f()\) neither directly nor indirectly. (For the overriding relationship involving methods in two classes, the relevant access mode is the one from the base class.) The most derived class that contains a version of \(f()\) overriding \(A.f()\) is \(B\), as both \(A.f()\) and \(B.f()\) have package access mode, and both \(A\) and \(B\) are in the same package. Consequently, the first method call will execute \(B.f()\).

The second call, \(((A)\text{ new } C()).g()\), is more interesting. The only difference between \(f()\) and \(g()\) in the hierarchy is the access mode for the version inside \(B\): we have \(\text{void } f()\), but \(\text{protected } \text{void } g()\). This time \(C.g()\) is able to override \(A.g()\) indirectly, through \(B.g()\). More exactly, \(B.g()\) overrides \(A.g()\) because both have package access mode and are in the same package, and \(C.g()\) overrides \(B.g()\) because the latter has \(\text{protected}\) access mode, and \(C\) is derived from \(B\).

This example illustrates that it is insufficient to consider just the static type and the dynamic type of the target, when resolving method calls. It is required to analyze the whole inheritance chain between the two, taking into account access modifiers and packages along the way.

**K-Java Rules.** During Lookup Method Reference step the construct `methodInfo()` is rewritten into a method reference `methodRef()` (Figure 6.18).

A method reference has two arguments: the method signature and the defining class. It is also subsorted to `KResult`, for the purpose of some context rules later on.

The first rule is for static methods. Here the defining class is already known (DecC). In addition, the qualifier is no longer needed and is replaced with `nothing::void`, for an uniform representation of all static method calls.

The second rule is for instance private methods, for which qualifier is a non-null object reference. Here `methodRef` is computed the same way as in the previous rule.

The third rule is for instance methods for which qualifier was evaluated to `null`. In this case method invocation can no longer proceed and a `NullPointerException` is thrown.
SYNTAX  \[\text{MethodRef} ::= \text{methodRef}\left(\text{Signature, ClassType}\right)\]

SYNTAX  \[\text{KResult} ::= \text{MethodRef}\]

SYNTAX  \[\text{MethodName} ::= \text{MethodRef}\]

RULE \textbf{INVoke-methodInfo-static}

\[
\begin{array}{l}
\text{nothing::void} . \\
\text{methodInfo(Sig:Signature, _, class, DecC:ClassType, static, _)}(\_)
\end{array}
\]

\[
\frac{
\text{methodRef(Sig, DecC)}
}{
\text{methodInfo(Sig, _, class, DecC:ClassType, static, _)}(\_)
}\]

RULE \textbf{INVoke-methodInfo-instance-private}

\[
\begin{array}{l}
(\text{objectRef(, _)} :: -) . \\
\text{methodInfo(Sig, _, class, DecC:ClassType, instance, private)}(\_)
\end{array}
\]

\[
\frac{
\text{methodRef(Sig, DecC)}
}{
\text{methodInfo(Sig, _, class, DecC:ClassType, instance, private)}(\_)
}\]

RULE \textbf{INVoke-methodInfo-instance-on-null}

\[
\begin{array}{l}
\text{null :: -. methodInfo(, _, _, _, _, instance, _)}(\_)
\end{array}
\]

\[
\frac{
\text{throw new NullPointerException( null :: String )}
}{
\text{methodInfo(, _, _, _, _, instance, _)}(\_)
}\]

RULE \textbf{INVoke-methodInfo-instance-protected-or-public}

\[
\begin{array}{l}
(\text{objectRef(, DC:ClassType)} :: -) . \\
\text{methodInfo(Sig, _, _, _, instance, Acc:AccessMode)}(\_)
\end{array}
\]

\[
\frac{
\text{methodRef(Sig, DefC)}
}{
\text{methodInfo(Sig, _, _, _, instance, Acc:AccessMode)}(\_)
}\]

\[
\langle DC \text{classType} (\_ \text{Sig} \mapsto \text{DefC:ClassType} \_ ) \text{methods}
\]

\[
\text{REQUIRES Acc} = \_\_\_\_\text{protected} \lor \text{Bool Acc} = \_\_\_\_\text{public}
\]

Figure 6.18: Lookup method reference — syntax and rules
The last rule in the figure covers protected and public access modes. Here the defining class DefC may also be computed in one step. As we mentioned earlier, for these access modes the defining method is the one visible from the qualifier dynamic type DC, thus it is referred in the \( \langle \rangle \) methods associated with DC. We cannot use the method declaration class from methodInfo here (e.g. the 4th argument), because when the qualifier static type is an interface, the 4th argument of methodInfo is left empty by the previous step (last rule in Figure 6.16).

The semantics for package access mode is defined in Figure 6.19. The first rule rewrites the methodInfo term into the auxiliary function lookupPackageMethod. This function has three arguments: method signature, the chain of classes between qualifier static class SC and dynamic class DC (computed by getClassChain), and the placeholder \( \cdot \) for the method definition class, yet to be computed.

The context rule that follows heats lookupPackageMethod to the top of computation.

The remaining rules implement the logic of lookupPackageMethod. The chain of classes is traversed not bottom-up, from derived to base, as would be expected from the JLS specification, but top-down, from SC to DC.

The rule lookupPM-first initializes lookup result with the method defining class DefC visible from the first class in the chain CurrentC. The two classes may be different.

The rule lookupPM-new-method updates this result to a lower class in the chain, if a new version of method with signature Sig is found, in case the old method is accessible. Accessibility semantics is given in Figure 6.20.

The rule lookupPM-no-new-method covers the case when the current class in the chain has no methods with Sig defined, it just advances the class chain to the next position.

The last rule matches when the class chain is entirely consumed, in this case the lookup procedure is over and methodRef is computed based on the last DefC found so far.

An example evaluation sequence for lookupPackageMethod() is given in Figure 6.21.

Although it would be possible to define this portion of semantics by traversing the chain bottom-up, such definition in \( K \) would be extraordinary.
RULE `INVOCER-METHODINFO-INSTANCE-PACKAG`E
  ( objectRef (—, DC:ClassType) :: SC:ClassType ) .
  
  methodInfo ( Sig:Signature, SC, class, —, instance, package )
      lookupPackageMethod ( Sig, getClassChain ( SC, DC ), K )

CONTEXT
  — . ☐:LookupPM( — )

SYNTAX  LookupPM ::= lookupPackageMethod ( Signature, ClassTypes, K ) [strict(2,3)]

RULE lookupPM-first
  lookupPackageMethod ( Sig, CurrentC:ClassType, Cs:ClassTypes , K )
      ⟨ ⟨ CurrentC ⟩ classType { ⟨⁻ Sig ↦ DefC:ClassType … ⟩ methods ⟩ ⟩

RULE lookupPM-new-method
  lookupPackageMethod ( Sig, CurrentC, Cs, OldDefC )
      ⟨ ⟨ CurrentC ⟩ classType { ⟨⁻ Sig ↦ DefC:ClassType … ⟩ methods ⟩ ⟩
          if ( isAccessible ( OldDefC, Acc, CurrentC )
              CurrentC else OldDefC

RULE lookupPM-no-new-method
  lookupPackageMethod ( Sig, CurrentC, Cs , — )
      ⟨ ⟨ CurrentC ⟩ classType { ⟨⁻ Sig ↦ DefC:ClassType … ⟩ methods ⟩ ⟩
          requires CurrentC ≠ K DefC

RULE lookupPM-end
  lookupPackageMethod ( Sig:Signature, `ClassTypes, DefC:ClassType )
      methodRef ( Sig, DefC )

Figure 6.19: Lookup method reference for package access mode
SYNTAX   \[ KItem ::= \text{isAccessible}(\text{ClassType}, \text{AccessMode}, \text{ClassType}) \]

RULE

\[
\text{isAccessible}(\_\_, \text{public}, \_\_)
\]

\[
\begin{array}{c}
\text{true}
\end{array}
\]

RULE

\[
\text{isAccessible}(\_\_, \text{protected}, \_\_)
\]

\[
\begin{array}{c}
\text{true}
\end{array}
\]

RULE

\[
\text{isAccessible}(\text{BaseC:ClassType}, \text{package}, \text{SubC:ClassType})
\]

\[
\begin{array}{c}
\text{getPackage}(\text{getTopLevel(\text{BaseC})}) == \text{getPackage}(\text{getTopLevel(\text{SubC})})
\end{array}
\]

RULE

\[
\text{isAccessible}(\_\_, \text{private}, \_\_)
\]

\[
\begin{array}{c}
\text{false}
\end{array}
\]

Figure 6.20: Syntax and rules for \text{isAccessible}()
<table>
<thead>
<tr>
<th>Configuration Fragment</th>
<th>Matched Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨lookupPackageMethod(g(), (a.A, a.B, b.C), 'K') ⟩<em>k \langle A \rangle</em>{classType} \langle \rightsquigarrow g() \mapsto A \rangle \langle \text{methods} \rangle_k</td>
<td>LOOKUPPM-FIRST</td>
</tr>
<tr>
<td>⟨lookupPackageMethod(g(), (a.B, b.C), a.A) ⟩<em>k \langle a.A \rangle</em>{classType} \langle g() \rangle_{methodSignature} \langle \text{package} \rangle_{methodAccessMode} \langle \text{class} \rangle \langle a.B \rangle_{classType} \langle \rightsquigarrow g() \mapsto a.B \rangle \langle \text{methods} \rangle \langle \text{class} \rangle</td>
<td>LOOKUPPM-NEW-METHOD, isAccessible returns true</td>
</tr>
<tr>
<td>⟨lookupPackageMethod(g(), (b.C), a.B) ⟩<em>k \langle a.B \rangle</em>{classType} \langle g() \rangle_{methodSignature} \langle \text{protected} \rangle_{methodAccessMode} \langle \text{class} \rangle \langle b.C \rangle_{classType} \langle \rightsquigarrow g() \mapsto b.C \rangle \langle \text{methods} \rangle \langle \text{class} \rangle</td>
<td>LOOKUPPM-NEW-METHOD, isAccessible returns true</td>
</tr>
<tr>
<td>⟨methodRef(g(), (\text{classTypes}, b.C)) ⟩_k</td>
<td>LOOKUPPM-END</td>
</tr>
</tbody>
</table>

Figure 6.21: The evaluation sequence of \texttt{lookupPackageMethod()} for method \texttt{f()} from Figure 6.17

inefficient performance-wise. Yet, overriding, just as the whole semantics, is thoroughly tested for conformance.

### 6.8.5 Actual Method Invocation

As a result of the previous steps, the method invocation expression reaches the form \texttt{Qual.methodRef(Sig, DefC)(Args)}. At this point, the main rule for method invocation is applied (Figure 6.22). This rule only matches when the method has a body (as checked by the side condition), e.g. is not a native method. Native methods, such as those presented in the next section, are implemented through specialized K rules.

This rule performs the following operations:

- The rest of computation (\texttt{RestK}) and the current content of \texttt{methodContext} are saved to a new entry (or frame) on top of the cell \texttt{stack}. This data will be used to restore the current computation context by the rules for \texttt{return}.
\textbf{RULE Invoke-methodRef}

\[
\begin{array}{c}
\text{Qual:KResult . methodRef (Sig, DefC)(Args:TypedVals) } \leadsto \text{RestK} \\
\text{staticInit (DefC) } \leadsto \text{initParams (Params, Args) } \leadsto \text{Body } \leadsto \text{return ;} \\
\text{List } \leadsto \text{stack} \\
\text{sl (RestK, MethodContext) } \leadsto \text{MethodContext:Bag} \\
\langle \text{Map}\rangle_{\text{env}} \langle \text{DefC}\rangle_{\text{crntClass}} \langle \text{getOId (Qual)}\rangle_{\text{location}} \langle \text{DefC}\rangle_{\text{classType}} \langle \text{Sig}\rangle_{\text{methodSignature}} \langle \text{Params:Params}\rangle_{\text{methodParams}} \langle \text{Body:K}\rangle_{\text{methodBody}} \\
\text{REQUIRES Body } \neq_{K} (\ ; \ )
\end{array}
\]

\textbf{SYNTAX} \quad KItem ::= \text{initParams (Params, TypedVals )}

\textbf{RULE initParams}

\[
\begin{array}{c}
\text{initParams (T:Type X:Id, RestP:Params , ( TV:TypedVal, RestV:TypedVals )) } \\
\text{T X ; } \leadsto \text{X = (T) TV ; } \leadsto \text{initParams (RestP, RestV)}
\end{array}
\]

\textbf{RULE initParams-end}

\[
\begin{array}{c}
\text{initParams ('Params, 'TypedVals ) } \\
\text{'}_{K}
\end{array}
\]

Figure 6.22: Rules for actual method invocation
• The new method context is initialized with:

  - An empty local variable environment ($\{\}_{\text{env}}$).
  - The defining class of the invoked method ($\langle\langle\langle \text{DefC} \rangle\rangle\rangle_{\text{crntClass}}$).
  - The current object location, computed by auxiliary function $\text{getOid}(\text{Qual})$ as:
    * The location of the qualifier, if qualifier is an object.
    * No location, if qualifier is $\text{nothing}::\text{void}$ (for static methods).

• Current computation is rewritten into a sequence of four terms:

  - Static initialization of the target class, in case it was not initialized already (function $\text{staticInit()}$). Repeated calls of this function have no effect.
  - Initialization of the parameters (function $\text{initParams()}$, explained below).
  - The method body.
  - A return statement with no arguments. It is used when method end is reached without encountering an explicit return.

The function $\text{initParams()}$ is also defined in Figure 6.22. It takes as arguments two lists — the list of parameter declarations and the list of argument values. The first rule processes the first parameter declaration in the list. The parameter with name $X$ of type $T$ is rewritten into a local variable declaration $T X$; followed by an assignment expression that initializes $X$ with the argument value. The third term in the RHS of the rewrite is $\text{initParams}$ with the processed parameter and argument removed.

When all parameters are processed, both arguments of $\text{initParams}$ become empty lists. In this case the third rule from the figure applies, dissolving the $\text{initParams}$ term.

6.9 Multithreading and Synchronization

K-Java has basic support for threads. First is the class Thread with methods $\text{start()}$, $\text{join()}$ and $\text{interrupt()}$. We also support thread synchronization through the synchronized statement (JLS §14.19) and the
synchronized modifier for methods (JLS §8.4.3.6). For more advanced usage we support threading-related methods from class Object: wait(), notify(), notifyAll(). In this section we present rules for most of the enumerated features. For the full semantics please see Appendix Section A.14.

6.9.1 Thread Startup and Termination

In Java, threads are started by calling the method Thread.start() on a thread object. The thread object has to be constructed in advance, with the constructor argument an implementation of the interface Runnable. This interface declares the method run() — the code that has to be executed in the new thread.

The logic of thread creation is implemented partially through Java code and partially through rules. More generally, this pattern was adopted for a small set of classes from JDK that we support, that also contain native methods. For those classes, K-Java comes with its own version of Java code, adapted to be better interfaced with K. The implementation of native methods, such as Thread.start() is done through K rules.

For the class Thread, the Java code bundled with K-Java is given in Figure 6.23, upper part. When a thread is instantiated, its field tid (thread identifier) is initialized with a fresh value. Later, when Thread.start() is invoked on a thread instance, it invokes the K-Java internal method startImpl(tid), declared native. The purpose of having startImpl native, instead of start directly, is to pass the TId argument to the rules implementing thread startup.

The invocation of native methods proceeds mostly as for regular methods. K-Java first evaluates them up to a methodRef. In the last step, they need specialized rules to perform the actual method invocation.

The logic for startImpl() is defined by the first rule in Figure 6.23 (see Figure 6.1 for the configuration cells). This rule refers two threads: the thread that calls startImpl() and the newly created thread. The upper part of the rule matches the old thread. Here, the call to startImpl() is rewritten into nothing::void, the standard return value for the return type void.

The lower part of the rule rewriting an empty bag (\(\langle\langle\langle\rangle\rangle\rangle\)) into a \(\{\}\)thread — the newly-started thread. Note that only two subcells are specified, \(\{\}\)k and \(\{\}\)tid, although \(\{\}\)thread has many more. In K, when a container cell,
public class Thread implements Runnable {

    private static int nextTid = 1;

    private Runnable runnable;
    private int tid = nextTid++;

    public Thread(Runnable runnable) {
        this.runnable = runnable;
    }

    public void run() {}

    public void start() { startImpl(tid); }

    private native void startImpl(int tid);
...
}

Figure 6.23: Code and rules for thread creation and thread termination
such as $\{\text{thread}\}$ is created, only a part of the subcells may be specified. The omitted cells will be automatically inferred by the configuration abstraction and initialized with their default values specified in the configuration. All the cells in a $\mathbb{K}$ definition have default values; for dynamic K-Java they are given in the Appendix Section [A.1] in the configuration figure.

Back to our rule, for the newly created thread the $\{\text{tid}\}$ is initialized with the parameter of $\text{startImpl()}$ method. The cell $\{\text{k}\}$ is initialized with the method call $\text{typedLookup(OId).Runnable.run();}$, where $\text{typedLookup(OId)}$ will evaluate to a reference to the started thread object. Note that $\text{Runnable}$ is a field of $\text{Thread}$ (Figure 6.23).

**Thread termination.** In K-Java a thread is terminated when cell $\{\text{k}\}$ becomes empty (second rule in Figure 6.23). The rule dissolves the $\{\text{thread}\}$ and registers the $\text{Tid}$ in the set of terminated threads.

### 6.9.2 Thread Synchronization

In Java, thread synchronization is achieved through $\text{synchronized}$ statement and methods with $\text{synchronized}$ modifier. Since synchronized methods are trivially desugared into regular methods with body wrapped into a $\text{synchronized}$ statement, below we only discuss the $\text{synchronized}$ statement.

The $\text{synchronized}$ statement has two components: the expression evaluated into a reference to the monitor object, and the executed block. When the execution of a synchronized block begins, the lock on the monitor object is acquired, thus preventing other threads to synchronize on the same object. If a thread reaches $\text{synchronized}$ statement when the lock on the target object is already held by another thread, this thread has to wait until the lock is released.

The rules for thread synchronization are given in Figure 6.24 (see Figure 6.1 for the configuration cells). First, the expression of the $\text{synchronized}$ statement is evaluated by strictness. Next, the first rule is matched at the beginning of the $\text{synchronized}$ statement, if the monitor is not held by any thread. This condition is checked by the side condition of the rule. In this rule, two cells responsible for thread synchronization are updated: the thread-local cell $\{\text{holds}\}$ and the global cell $\{\text{busy}\}$. The cell $\{\text{hold}\}$ contains a map from the acquired locks to the number of times the lock was acquired for each
SYNTAX \[ Stmt ::= \textbf{synchronized} \ ( \ Exp \ ) \ Block \ [\textit{strict}(1)] \]

RULE \textbf{Synchronized-first-time}
\[
\begin{aligned}
\text{synchronized} \ ( \ \text{objectRef} \ ( \ OId : \textit{Int}, \ - ) :: \ - ) \ S : Stmt \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\end{aligned}
\]

REQUIRES \[ \neg \textit{Bool} \ ( \ OId \ \text{in} \ \text{Busy} ) \]

RULE \textbf{Synchronized-nested}
\[
\begin{aligned}
\text{synchronized} \ ( \ \text{objectRef} \ ( \ OId : \textit{Int}, \ - ) :: \ - ) \ S : Stmt \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\text{try} \ S \ \text{finally} \ \{ \ \text{releaseLock} \ ( \ OId ) \} \\
\end{aligned}
\]

RULE \textbf{releaseLock}
\[
\begin{aligned}
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\end{aligned}
\]

RULE \textbf{releaseLock-complete}
\[
\begin{aligned}
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\text{releaseLock} \ ( \ OL : \textit{Int} ) \\
\end{aligned}
\]

Figure 6.24: Rules for synchronized statement
object identifier. The cell \( \langle \rangle_{\text{busy}} \) contains the set of objects for which the lock is acquired by any thread. In the computation, the \textit{synchronized} statement is rewritten into a \textit{try/finally} statement that executes the synchronized block \( S \), while ensuring the lock is released at the end, regardless how \( S \) execution terminates.

If the monitor is held by another thread, then no rule can match this computation, and the \( \mathcal{K} \) term rewriting engine will advance the state by evaluating other threads. Thus, at some point, if the program is not stuck in a deadlock, the monitor object will be unlocked, and the rule discussed above will be able to match.

The second rule in the figure is very similar to the first, matched when the thread tries to synchronize on an object while already holding the lock on the same object. In this case the counter of acquired locks in \( \langle \rangle_{\text{holds}} \) is simply incremented.

The role of the last two rules in the figure is to release the lock at the end of the synchronization statement, symmetrically to the way it was acquired by the first two rules.

### 6.9.3 Methods \texttt{wait()} and \texttt{notify()}

Below we discuss the key rules for methods \texttt{wait()} and \texttt{notify()}. Both methods have to be called from a \textit{synchronized} block that holds the monitor (lock) of the target object. When \texttt{wait()} is called, the thread releases the lock on target object and blocks until another thread calls \texttt{notify()} on the same object. When \texttt{notify()} is called, the waiting thread does not wake up immediately, but only after the notifying thread exited the \textit{synchronized} block. When waking up, the thread re-acquires the lock on the target object. When there are multiple threads waiting on the same monitor object, \texttt{notify()} will non-deterministically wake up one of them. Finally, if either \texttt{wait()} or \texttt{notify()} is called in a state where the current thread does not hold the appropriate monitor, an exception is thrown.

The rules are presented in Figure 6.25. The first rule \texttt{OBJECT-WAIT} corresponds to the invocation of \texttt{wait()}. The upper part of the rewrite inside \( \langle \rangle_k \) is the final form of the method call to \texttt{wait()} with no arguments, on an object with object identifier \texttt{OId}. The \texttt{OId} serves both as identifier in the \( \langle \rangle_{\text{store}} \) and as synchronization key. This rule replaces the method call
Figure 6.25: Key rules for Object.wait() and Object.notify()
expression in cell $\langle k \rangle$ with \texttt{waitImpl()}—an auxiliary construct used later, to exit from the waiting state. The id of the current thread (\texttt{TId}) has to be registered in the set in $\langle waitingThreads \rangle$. The cell $\langle holds \rangle$ attached to each thread stores the number of times the current thread acquired the lock on each object. Here we use it to make sure that the current thread acquired the lock at least once (see the side condition). We also have to delete \texttt{OId} from the set $\langle busy \rangle$, to release the ownership of the monitor during the waiting state.

The second rule \texttt{OBJECT-notify} is the starting rule for \texttt{notify()}. The side condition ensures that the current thread holds the monitor on the target object. The actual logic of \texttt{notify()} is delegated to \texttt{notifyImpl()}. The construct \texttt{notifyImpl()} requires two rules for two cases — the case when there is at least one thread waiting on the target object, and the case when there is none.

Rules \texttt{NOTIFYImpl-someone-waiting} and \texttt{NOTIFYImpl-no-one-waiting} are for the first case. If there is a thread waiting on the current object, then the object identifier \texttt{OId} will be present among the map values of $\langle waitingThreads \rangle$. By deleting the whole entry associated to that value we enable the waiting thread to proceed. In the second rule, if there is no thread waiting for this object, then the execution of \texttt{notifyImpl()} ends with no effect. In both rules the RHS of the rewrite is \texttt{nothing::void}, because in K-Java every function, even a native void one, has to return a value.

At this stage, after a call to \texttt{notify()}, the rule for \texttt{waitImpl()} could match inside another thread (rule \texttt{waitImpl-main}). The rule checks in its side conditions that the current thread identifier, \texttt{TId}, is not among the waiting threads anymore. It also checks that the target object, represented by \texttt{OId}, is not busy. This is required because the thread exiting from waiting state has to reacquire the monitor on the target object. Finally, the rule has to make sure that the thread was not interrupted while it was waiting. Otherwise another rule will match and will throw the appropriate exception.

We only presented the key rules above, corresponding to the most common execution scenario. The corner cases, such as illegal calls that should result in various exceptions, are fully supported and tested, included in the appendix.
Chapter 7

Applications

Here we show how K-Java together with builtin \( \mathbb{K} \) tools can be used to explore multi-threaded program behaviors. The first application is state space exploration and the second is LTL model-checking.

7.1 State Space Exploration

The next simplest way to use K-Java besides execution is state space exploration. When a program is run with the \( \mathbb{K} \) runner and option \(--\text{search}\), the tool outputs all of the possible executions for the program, exposing any possible non-deterministic behavior. This capability was successfully used in the semantics of C \[18\] to expose the non-determinism of expression evaluation. While single-threaded Java is deterministic, threads bring non-determinism. By running a multi-threaded Java program in search mode, we can produce all its interleaving behaviors.

Additionally, the option \(--\text{pattern}\) allows us to filter the search results according to a pattern. This feature may be effectively used, for example, to detect deadlocks. In K-Java, the cell \( \{\}_{\text{thread}} \) is dissolved when the corresponding thread finishes its execution. Consequently, we can detect successful execution paths by using the pattern \( \{.\}_{\text{threads}} \). The pattern will match when there are no threads remaining. Conversely, the pattern \( \{\neg\}_{\text{thread}} \) would match the final states where at least one thread did not finish its execution, e.g. a deadlock. We successfully used this approach to detect the deadlock in the Dining Philosophers problem.

7.2 LTL Model-Checking

While state space search might be used to test some programs, \( \mathbb{K} \) offers a more powerful capability for exploring non-deterministic behavior. Specifically, \( \mathbb{K} \)
provides linear temporal logic (LTL) model-checking capabilities through its Maude \[14\] backend. In this section we show how K-Java can be seamlessly used to verify LTL properties of multi-threaded applications.

Consider the program in Figure 7.1 (a modified version of \[1\]). The program contains a blocking queue – a classical producer-consumer example with one producer and one consumer thread. The inner class BlockingQueue contains two methods: put() and get(). The methods are synchronized, designed to be called from a multi-threaded environment. When put() is called on a full queue, the calling thread has to wait until some other thread dequeues an element. Similarly, the consumer thread calling get() has to wait when the queue is empty. The producer (the main thread) calls put() four times, while the consumer (anonymous thread inside main()) calls get() the same four times. Aside from demonstrating multi-threading capabilities, this small program illustrates many other features of Java: expressions with side effects, exception handling, arrays, static and instance methods, inner and anonymous classes. Running state space exploration for the example above correctly produces all eight expected interleavings of the output text (0 represents a call to put(), 1 – a call to get()):

00101011 00101101 00110011 00110101
01001011 01001101 01010011 01010101

The implementation of BlockingQueue contains a deliberate, subtle problem. The wait() call is within an if inside both get() and put(), thus is executed at most once. This is actually a correct behavior if we have just one producer and one consumer\[1\] but leads to problems when the number of threads is at least three. Above, the only way a thread waiting on line labeled '3' could be awakened is from a call to notify() from method put(), line labeled '2'. Since at the end of the method put() we have at least one element in the queue, method get() can safely extract an element.

In a scenario with one producer and two consumers the thread waiting on '3' could be awakened by a call to notify() from either '2' or '4'. If the thread executing get() was awakened by '2' (another get()), that other get() could have actually extracted the last element from the queue, thus rendering the queue empty. Unaware of the queue state, the freshly-awakened thread will execute the body of get() and will extract a non-existing element.

\[1\]We do not consider spurious wakeups here.
public class QueueTest {
    static class BlockingQueue {
        int capacity = 2;
        int[] array = new int[capacity];
        int head=0, tail=0;
        synchronized void put(int element) throws InterruptedException {
            if (tail-head == capacity) {
                wait(); // 1
            }
            array[tail++ % capacity] = element;
            System.out.print(0);
            notify(); // 2
        }
        synchronized int get() throws InterruptedException {
            if (tail-head == 0) {
                wait(); // 3
            }
            int element = array[head++ % capacity];
            System.out.print(1);
            notify(); // 4
            return element;
        }
    }
    public static void main(String[] args) throws Exception {
        final BlockingQueue queue = new BlockingQueue();
        new Thread() {
            public void run() {
                for(int i=0; i<4; i++) {
                    try {
                        queue.get();
                    } catch (InterruptedException e) {} 
                }
            }
        }.start();
        for(int i=0; i<4; i++) {
            queue.put(i);
        }
    }
}

Figure 7.1: A two-threaded blocking queue
from the queue. To eliminate this possibility, we need to replace the if statements with while above both at '1' and '3'. A programmer might make such a subtle mistake, when first designing the queue for a two-threaded usage, and then extending the context to more threads. It is difficult to expose this bug through traditional testing. We executed the incorrect three-threaded program a dozen times with the JVM; it never picked the buggy path.

Next, we would like to expose the problem above with LTL. In this implementation of queue tail is incremented each time an element is added to the queue, and head is incremented each time an element is extracted. They are never decremented. Since it is not possible to extract more elements than are added, a basic invariant of the queue is head <= tail. This is the exact invariant that is violated by the example above in the three-threaded scenario.

The LTL model checking capability of the K framework allows us to find the bug by model checking the following LTL formula:

\[ \Box (\text{this instanceof BlockingQueue} \implies \text{this.head} \leq \text{this.tail}) \]

The property was correctly proven true for the example in Figure 7.1 but was proven false for the three-threaded version of the same program. When we corrected the queue implementation, the model checker proved the formula correct. The version of the program in Figure 7.1 took 15 seconds to model check, produced 72 transition states and roughly two million rewrites.

**Transition sets** In order to enable the LTL model-checking support, we had to label the rules that exhibit non-deterministic behavior as transitions. K model-checker explores all execution orders only for such transition rules. We actually decided to have two sets of transitions — the full set and the synchronized set. The full set includes all of the rules related to threading, field assignment and lookup, as well as printing to the standard output. This set of transitions enables us to precisely test the behavior of any multi-threaded code. It was used for testing the semantics of multithreading. The synchronized set of transitions do not include the rules related to variable access and writing to standard output. This set is suitable for programs that are known in advance to be correctly synchronized. E.g. programs that access shared variables and resources only inside proper synchronized blocks. We used the second set to model check the example above. Together the two sets of transitions allow
for a choice between precision and performance. Transition attributes were removed from rules in Section 6.9 but are preserved in Appendix A.

Similar formal program analysis capabilities were demonstrated within the semantics of C [18] and PHP [20].
Chapter 8

Testing

Testing K-Java took almost half of the overall development time. Here we describe our testing efforts, which resulted in what could be the first publicly available conformance test suite for Java.

8.1 The Quest for a Test Suite

Virtually all recent executable language semantics projects \[18, 20, 4, 41\] used an external test suite for validation. Naturally, we tried to do the same. The official test suite for Java, targeting both the language and the class library, is Java Compatibility Kit (JCK) \[38\] from Oracle. JCK is not publicly available. Instead Oracle offers free access for non-profit organizations willing to implement the whole JDK, i.e., both the language and the class library. After a laborious application, Oracle rejected our request.

We also explored unofficial test suites. Jtreg, part of OpenJDK \[37\], is a regression test suite for the class library, but not for the language. Another test suite is Mauve \[35\], containing tests for classes and for the compiler, but not for the runtime. Tests targeting the compiler test its capability to distinguish between correct and incorrect Java programs, and to output the appropriate error message. Unfortunately, all these tests were unsuitable for our purpose.

There were actually two external test suites that we were able to use. One was the set of examples from ACM-Java \[48\] that we presented in Chapter 2. Another one was the list of examples from the book Java Precisely \[47\]. We used 44 out of 58 tests from ASM-Java and 63 out of 114 examples from Java Precisely. The programs that we could not use either illustrated compiler errors, or were not complete Java programs, or used classes with native functions that are not supported by K-Java.

These two sets of examples, while useful, were far from enough. Their
purpose was to illustrate Java, not to exercise every single corner case of the language. With no luxury of an available comprehensive test suite, we had no choice but to develop our own.

### 8.2 Test Development Methodology

When writing our tests we followed the Test Driven Development (TDD). The main principle of TDD is to write tests before implementing the actual feature under test. It was advantageous to use TDD for K-Java for two reasons. First, K-Java has a complete and final specification—JLS. Consequently, our tests are not expected to change as a result of changes in the specification. Second, tests for K-Java are self-contained Java programs, they do not depend on any part of the system under test (K-Java) to be written.

For every test, we compared the output produced by K-Java with the output produced by JDK. When developing a new feature, we followed the following steps. First we tried to cover all corner cases of the feature under test in isolation. For example, the first non-terminal of the `for` statement might be a list of variable initializers. In such case we will include in our tests statements with zero, one or more initializers. Second, we would define the new feature in the simplest way possible to pass all the tests. Sometimes, after inspecting the implementation, we would identify some corner cases that were not captured by the tests. We would add additional tests for such cases. Thus our methodology was a combination of white-box and black-box testing. In addition to the steps above, we wrote tests for each combination of language features whenever we thought that the two features may possibly unexpectedly interact.

We followed the development of Java starting from low-level features such as literals, expressions, statements, towards the higher level features. The order of development is approximately reflected by the order in which tests were written, which can be found in K-Java public repository [5]. We aimed at testing every detail specified in JLS; e.g., to test the precise order of execution of subexpressions inside an expression, we intentionally used subexpressions with side-effects and verified the correct order of evaluation by observing the correct order in which the side effects occurred.

Some features depended on other features to be properly tested. For
example, in order to test the precise static type of various expressions we used
method overloading. For example, if an expression \( e \) has to return type \( A \), but
a plausible erroneous semantics could also produce \( B \), we tested the correct
choice by calling \( f(e) \), where \( f \) was an overloaded method that could accept
an argument of type \( A \) or \( B \). In all such cases we were careful to postpone
the exhaustive testing of the freshly developed feature, and to write the tests
later once all prerequisites are available.

Eventually, we produced a test suite consisting of 840 tests.

**Multithreading.** To test multi-threading we used state space exploration.
We designed a test suite comprised of 28 programs explicitly aiming at covering
all the behaviors of all the supported multi-threaded language constructs. For
each program, we first produce all the possible solutions using \( K \). Then we
compare the number of solutions with the expected, manually determined
one.

**Later changes, testing ASM-Java and JavaFAN.** When we first tried
to execute ASM-Java and JavaFAN over our test suite, the majority of the
tests unexpectedly failed. It was because some very basic feature, used in
most of the tests, was not supported. For example ASM-Java did not support
addition between a string and a boolean. JavaFAN did not support escape
characters, and we were not able to use "\n" in our tests for this reason.
To overcome this problem, we inspected our tests and eliminated the most
common causes of failure, when they were not the actual feature under test.
This way we were able to produce the results reported in Chapter 2.
Chapter 9

Conclusion

We have presented K-Java, which to our knowledge is the first complete formal semantics of Java. The semantics has been split into a static and a dynamic semantics, and the static semantics was framed so that its output is also a valid Java program. This way, it can seamlessly be used as a frontend in other Java semantics or analysis tools. As a side contribution we have also developed a comprehensive conformance test suite for Java, to our knowledge the first public test suite of its kind, comprising more than 800 small Java programs that attempt to exercise all the corner cases of all the language constructs, as well as non-trivial interactions of them.

The skeptical reader may argue that there is no such thing as a ‘complete’ semantics of a large language like Java, because it is always possible to miss a feature or, worse, an interaction of features. While this is true in principle, we mention that completeness of the semantics was our major objective from the inception of this project, and that we have very carefully considered all possible interactions of features that were explicitly discussed in the JLS or that we could think of. Since there is no other attempt to completely formalize Java that we are aware of in order to formally compare with K-Java, due to all the above we believe that it is fair to claim that K-Java is the first complete formal semantics of Java.

9.1 Statistics

The size of K-Java can be roughly assessed based on the statistics below:
<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
<th>Common</th>
<th>Lib</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLOC</td>
<td>3112</td>
<td>2035</td>
<td>539</td>
<td>497</td>
<td>6183</td>
</tr>
<tr>
<td>CLOC</td>
<td>857</td>
<td>1189</td>
<td>383</td>
<td>98</td>
<td>2527</td>
</tr>
<tr>
<td>Size (KB)</td>
<td>168</td>
<td>139</td>
<td>52</td>
<td>22</td>
<td>381</td>
</tr>
<tr>
<td>N cells</td>
<td>31</td>
<td>36</td>
<td>28</td>
<td>–</td>
<td>95</td>
</tr>
<tr>
<td>N rule</td>
<td>497</td>
<td>347</td>
<td>183</td>
<td>47</td>
<td>1074</td>
</tr>
<tr>
<td>N aux</td>
<td>111</td>
<td>83</td>
<td>79</td>
<td>9</td>
<td>282</td>
</tr>
</tbody>
</table>

The rows represent source lines of code (SLOC), comment lines of code (CLOC), total files size, number of cells, number of rules, number of auxiliary syntactic constructs. The columns represent the static semantics, the dynamic semantics except class library, modules common for both static and dynamic semantics, class library (including multi-threading) and the total size.

Overall, Java 1.4 has 186 syntactic productions (JLS[27, §18 ]), K-Java covers all of them. The whole semantics took approximately 2 man-years to develop, including the tests.

### 9.2 A Discussion on $K$

In this section we subjectively evaluate $K$, with K-Java as an example, in two of its secondary roles: suitability for formal definitions of real-life languages, and suitability for programmer references.

**Suitability for Formal Definitions**  With K-Java, we have proved for the second time that $K$ can handle complete statically typed real-life language specifications. The first complete semantics with these properties was for C. Now we used the same techniques to produce a complete semantics of Java, which, being an object-oriented language, is a step further in complexity than the semantics of C. The great complexity of Java and the intricate rules for expression types and local classes forced us to take a new approach to modularity: to split the whole semantics into two autonomous definitions, the static and the dynamic one.

It was relatively straightforward to define the dynamic semantics. Here $K$-specific abstractions such as strictness or configuration abstraction helped us to reduce the number of rules and their size. Many design decisions were
already fleshed out, being adapted from toy \( \mathbb{K} \) definitions SIMPLE \[44\] and KOOL \[44\].

However, with the static semantics we were stepping on a less explored path. Strictness was less effective here, we had to rely more often on manually written heating and cooling rules. Cells were not flexible enough to express the state during elaboration phase, and we had to use auxiliary constructs instead, that do not provide configuration abstraction. And some of auxiliary constructs had their definition scattered across the semantics. As a result, static K-Java was less modular, with harder to read rules. For this reason we decided not to include any rules from the static semantics in the main body of this thesis. Now that our part in the project is over, we can identify a number of design decisions that could have been done better, leading to a potentially simpler static semantics. Thus, the entanglement of static K-Java is probably caused by a combination of poor design decisions and \( \mathbb{K} \) abstractions being less suitable for defining transformations. We expect this aspect to be better evaluated once more statically typed languages that require extensive preprocessing get their semantics defined in \( \mathbb{K} \).

**Suitability for Programmer References** One of the long-term goals of \( \mathbb{K} \) was to enable formal language semantics that are simple enough to be included in the official specifications of the respective languages. It is customary now for an official language specification to include the formal definition of syntax, as is the case, for example, for JLS\[27, \S18 \]. It is not the case for semantics, however. For a formal semantics, to be included in an official specification, it has to be simple enough to be readable by programmers. \( \mathbb{K} \) semantics for simple languages, such as SIMPLE or KOOL are close enough to this goal. However, even for the dynamic K-Java alone we see two issues that would impair its usability by programmers. First issue is that K-Java rules are defined over AST. The issue will disappear naturally when K-Java will be defined over Java syntax.

The second, more serious difficulty is the presence of auxiliary constructs, such as \_::\_, objectRef, etc. They add an extra complexity that is not present in language specifications given as plain prose, such as JLS. On the other hand, such constructs may very well be inevitable. The specification of JavaScript \[15\] includes a pseudocode definition of the language, that also contains a lot of auxiliary constructs. Hence, we expect KJS \[39\] to be a
better use case to evaluate $K$ in this role.

### 9.3 Limitations

We do not define the Java Memory Model (JMM) that governs the allowed multi-threaded execution behaviors (§17). Instead, we define a sequentially consistent memory model with a global store and no caches, consistent with the JLS only if all fields are volatile. Defining JMM would be a significant effort on its own, relatively orthogonal to the present work. Also, JMM 1.4 was replaced by JMM 5, so is no longer actual. It has been demonstrated that $K$ allows for modular replacement of the memory model \[17, §4.2.6\]. Consequently, there is potential to implement JMM on top of K-Java in a modular fashion. So far, JMM was explored only on a core subset of Java \[34\], the present work allows defining JMM on full Java.

### 9.4 Future Work

An avenue for future work is extending K-Java with semantics for the latest versions of Java, starting with Java 5. Most of Java 5 features can be compiled into JVM 1.4 bytecode by using javac option `-target jsr14`: generics, varargs, for-each loop and autoboxing \[26\]. These features can be similarly defined in the static K-Java. Support in the dynamic K-Java would be required for enumerations and annotations. The best-known feature of Java 8 is lambda expressions. Lambdas could be desugared into anonymous classes implementing a functional interface. JDK uses a different strategy, yet the designers of Java suggest desugaring as an option \[25\]. Adding support for the latest Java Memory Model would allow a less-constrained reasoning about memory access in multithreading programs.

Another direction would be defining dynamic class loading. It has already been defined in K-Python \[28\], so we see no impediments in adapting it for K-Java. This would allow support for a much wider set of classes from JDK, consequently, for more real life programs.

At the moment of writing work is underway to convert K-Java from AST-based to Java syntax-based form, with the potential to enhance rules readability.
Finally, we are eagerly waiting K-Java to be migrated to the latest version of K, a task also currently performed by FSL group at UIUC. This should enable deductive verification of Java applications, similar to verification capabilities shown in [39]. Deductive verification requires, besides migration to the latest K, more user involvement than model checking, and would be a separate work on its own.
Appendix A

K-Java Dynamic semantics

A.1 Module CONFIGURATION-EXEC
The execution configuration consists from the top-level cell $\langle \rangle_T$, whose content is divided into three large cell groups. The group $\langle \rangle_{\text{threads}}$ contains one or multiple cells $\langle \rangle_{\text{thread}}$. Here is included the data related to the execution of a thread, such as the computation, stack, local environment. The second group is contained inside $\langle \rangle_{\text{classes}}$ that have multiple $\langle \rangle_{\text{class}}$ cells — all the content related to particular classes. The third group consists of the remaining top-level cells.

**Threads** Each execution thread has a corresponding $\langle \rangle_{\text{thread}}$. Inside a thread, $\langle K \rangle_k$ is represents the actual computation — the sequence of tasks to be executed. Next cells are $\langle \text{List}\rangle_{\text{stack}}$ — the method call stack, and $\langle \rangle_{\text{methodContext}}$. The $\langle \rangle_{\text{methodContext}}$ holds the local context of the executed method:

- $\langle \text{Map}[\text{Id} \rightarrow \text{Int}]\rangle_{\text{env}}$ — The map from local variable names to their location inside $\langle \rangle_{\text{store}}$.
- $\langle \text{ClassType}\rangle_{\text{crntClass}}$ — The current class, the type of this.
- $\langle \text{Int}\rangle_{\text{location}}$ — The location in memory store of the current object.

On the next line there are 3 cells related to multithreading:

- $\langle \text{Int}\rangle_{\text{tid}}$ — The thread identified, generated when the thread is created.
- $\langle \text{Map}[\text{Int} \rightarrow \text{Int}]\rangle_{\text{holds}}$ — A map from Object locations to integer numbers. Map keys are monitor objects on which the current thread has the locks. Map values hold the number of times the lock was acquired.
- $\langle \text{Bool}\rangle_{\text{interrupted}}$ — A boolean value signaling whether the current thread was interrupted.

**Classes** The cell $\langle \rangle_{\text{classes}}$ contains one $\langle \rangle_{\text{class}}$ for each class in the program, both for supported JDK classes and for user-defined classes. Cell $\langle \rangle_{\text{class}}$ has the following content:

- $\langle \text{ClassType}\rangle_{\text{classType}}$ — The fully qualified class name. This cell serves as identifier for the whole $\langle \rangle_{\text{class}}$, thus $\langle \rangle_{\text{classes}}$ may be seen as a map from class name to $\langle \rangle_{\text{class}}$ content.
- $\langle \text{ClassMetaType}\rangle_{\text{classMetaType}}$ — Indicates whether this type is class or interface. Both are referred hereafter as 'classes', making the distinctions only when necessary.
- $\langle \text{ClassPhase}\rangle_{\text{classPhase}}$ — Used during unfolding global phase (see below).
- $\langle \text{ClassType}\rangle_{\text{enclosingClass}}$ — For inner classes, the direct lexically enclosing class of this class. For top-level classes — no value.
- $\langle \text{ClassType}\rangle_{\text{extends}}$ — The base class.
- $\langle \text{Set } [ \text{ClassType} ] \rangle_{\text{implTrans}}$ — The transitive closure of implemented interfaces.
- $\langle \text{Map } [ \text{Signature} \rightarrow \text{ClassType} ] \rangle_{\text{methods}}$ — The map of accessible methods. Keys are method signatures, values are classes where methods are defined. Includes both methods declared within this class as well as methods inherited from base classes/ base interfaces.
• \( \langle Bag \rangle \) \text{methodDecs} — The collection of method declarations (\( \langle \rangle \) \text{methodDec} cells) in the current class. This cell contains only a subset of methods from \( \langle \rangle \text{methods} \), as the set of accessible methods from \( \langle \rangle \text{methods} \) also includes methods inherited from base classes/interfaces. Hence the need of two separate collections. Each \( \langle Bag \rangle \) \text{methodDec} contains the following data:
  - \( \langle Signature \rangle \) \text{methodSignature} — The method signature, acting as identifier of the \( \langle \rangle \text{methodDec} 
  - \( \langle \text{List[ Param ]} \rangle \) \text{methodParams} — The method parameters.
  - \( \langle K \rangle \) \text{methodBody} — method body
  - \( \langle AccessMode \rangle \) \text{methodAccessMode} — The method access mode.
  - \( \langle ContextType \rangle \) \text{methodContextType} — May be either static or instance.

• \( \langle K \rangle \) \text{instanceFields} — The list of instance field declarations, stored as a list of local variable declaration statements, without initializers. Used during object instantiation.

• \( \langle K \rangle \) \text{staticFields} — The list of static field declarations, in a similar format as \( \langle \rangle \text{instanceFields} \)

• \( \langle K \rangle \) \text{staticInit} — The list of static initializers and static field initializers concatenated into one block. The content of \( \langle \rangle \text{staticFields} \) and \( \langle \rangle \text{staticInit} \) is executed during class static initialization.

• \( \langle K \rangle \) \text{folded} — The initial representation of the class in AST format, during unfolding phase (see below).

• \( \langle \text{StaticInitStatus} \rangle \) \text{staticInitStatus} — The static initialization status of the class. May be either uninitialized, initializing or initialized. The content of this class is used to trigger static initialization of the class on first use.

• \( \langle \text{Map[ Id \mapsto Location ]} \rangle \) \text{staticEnv} — The map from static fields declared in this class to their locations in the store. Populated during static initialization. This cell along with \( \langle \rangle \text{staticInitStatus} \) are the only two cells of \( \langle \rangle \text{class} \) altered during program execution, during static initialization. The rest of the cells are populated before the execution.

The remaining cells The first three cells after \( \langle \rangle \) \text{classes} are important at the beginning of computation:

• \( \langle K \rangle \) \text{program} — The initial AST representation of the program, containing both core classes from JDK and user-defined classes.

• \( \langle K \rangle \) \text{mainClass} — The class containing the method \text{main()} — the starting point of execution. Loaded from a command line argument.

• \( \langle K \rangle \) \text{globalPhase} — Computation in the execution semantics may be in one of two global phases: Unfolding and Execution. The role of the unfolding phase is to convert the AST representation of the program stored in \( \langle \rangle \text{program} \) into the collection of cells \( \langle \rangle \text{classes} \). When this operation is complete, execution phase follows.

The next 4 cells represent the memory model of K-Java:

• \( \langle \text{Map[Location:Int \mapsto Value:TypedVal]} \rangle \) \text{store} — The program memory. Map from memory locations to actual values. For reference types only the reference is stored in this cell.
• \( \langle \text{Map} [\text{Location}: \text{Int} \mapsto \text{LocMetadata} ] \rangle \text{storeMetadata} \) — For each location in \( \langle \text{store} \rangle \), information whether that location is a local variable or field. Used for performance optimization of multi-threaded model-checking.

• \( \langle \text{Int} \rangle \text{nextLoc} \) — the next available memory location. Incremented after each allocation.

• \( \langle \text{Map} [\text{Type} \mapsto \text{TypedVal} ] \rangle \text{classLiteralsMap} \) — A map from types \( T \) to objects that represent the value of the expression \( T.\text{class} \).

• \( \langle \text{Bag} \rangle \text{objectStore} \) — Represents runtime state of all the instantiated objects. Multiple references to the same object are modeled through multiple entries in \( \langle \text{store} \rangle \) pointing to the same entry in \( \langle \text{objectStore} \rangle \). The cell contains multiple \( \langle \text{object} \rangle \) cells, each with the following content:
  - \( \langle \text{Int} \rangle \text{objectId} \) — A unique object identifier, pointed by each reference in \( \langle \text{store} \rangle \)
  - \( \langle \text{ClassType} \rangle \text{objectType} \) — The runtime type of the object.
  - \( \langle \text{Bag} \rangle \text{layer} \) — There could be multiple layer cells inside an \( \langle \text{object} \rangle \). Each layer represents a concrete class in the inheritance hierarchy, starting from \texttt{Java.lang.Object} and finishing with \( \langle \text{objectType} \rangle \). An object layer stores the following information:
    * \( \langle \text{ClassType} \rangle \text{layerClass} \) — The class represented by this layer.
    * \( \langle \text{Map} [\text{Id} \mapsto \text{Int}] \rangle \text{layerEnv} \) — A map from instance fields to their locations in the store.
    * \( \langle \text{TypedVal} \rangle \text{layerEnclosingObject} \) — A reference to the directly enclosing object if this object represents an instance inner class, or empty for other types of objects (top-level and static inner).

The following cells represent program I/O:

• \( \langle \text{List} \rangle \text{in} \) — The standard input, represented as pre-parsed list of elements of type Int or String.

• \( \langle \text{List} \rangle \text{out} \) — The standard output.

The next row contains global cells related to multithreading:

• \( \langle \text{Set} [\text{ObjLocation}: \text{Int}] \rangle \text{busy} \) — The set of busy objects. E.g. monitor objects for which there is a thread holding the monitor.

• \( \langle \text{Map} [\text{ThreadId}: \text{Int} \mapsto \text{OL}: \text{Int}] \rangle \text{waitingThreads} \) — The map from threads to monitor objects on which respective threads are blocked in a call to \texttt{wait()}. Used by the methods \texttt{Object.wait()}, \texttt{Object.notify()}, \texttt{Object.notifyAll()}.

• \( \langle \text{Set}[\text{ThreadId}: \text{Int}] \rangle \text{terminated} \) — The set of identifiers of terminated threads.

The rest of the global cells are used for debugging purposes.
A.2 Module EXPRESSIONS

The module defines expressions operating over primitive types and String.

Used to denote a placeholder for an expression type, until the proper, normalized type of an expression is computed. Never referred in the semantics. An expression containing a placeholder type should always be wrapped inside a cast expression. The cast will then replace tempType with the proper compile-time type.

**SYNTAX**

\[
\text{Type} ::= \text{tempType}
\]

## A.2.1 Boolean operators

**RULE**

\[
\text{true} :: \text{bool} \land \rightarrow \text{true}
\]

**RULE**

\[
\text{false} :: \text{bool} \land K \rightarrow K
\]

**RULE**

\[
\text{true} :: \text{bool} \land K \rightarrow K
\]

**RULE**

\[
\text{false} :: \text{bool} \land \rightarrow \text{false}
\]

**RULE**

\[
I_1 : \text{Int} :: \rightarrow I_2 : \text{Int} :: \rightarrow
\]

\[
(I_1 \lor \text{Int} I_2) :: \text{tempType}
\]

**RULE**

\[
B_1 : \text{Bool} :: \text{bool} \lor B_2 : \text{Bool} :: \text{bool}
\]

\[
(B_1 \lor \text{Bool} B_2) :: \text{bool}
\]

**RULE**

\[
I_1 : \text{Int} :: \rightarrow I_2 : \text{Int} :: \rightarrow
\]

\[
(I_1 \oplus \text{Int} I_2) :: \text{tempType}
\]
RULE

\[
\begin{align*}
\text{B1:Bool} &:: \text{bool} \lor \text{B2:Bool} :: \text{bool} \\
& (\text{B1 xorBool B2}) :: \text{bool}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{I1:Int} &:: — \land \text{I2:Int} :: — \\
& (\text{I1 } \land_{\text{Int}} \text{I2}) :: \text{tempType}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{B1:Bool} &:: \text{bool} \land \text{B2:Bool} :: \text{bool} \\
& (\text{B1 } \land_{\text{Bool}} \text{B2}) :: \text{bool}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{I1:Int} &:: — \Rightarrow \text{I2:Int} :: — \\
& (\text{I1 } \Rightarrow_{\text{Int}} \text{I2}) :: \text{bool}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{F1:Float} &:: — \Rightarrow \text{F2:Float} :: — \\
& (\text{F1 } \Rightarrow_{\text{Float}} \text{F2}) :: \text{bool}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{B1:Bool} &:: — \Rightarrow \text{B2:Bool} :: — \\
& (\text{B1 } \Rightarrow_{\text{Bool}} \text{B2}) :: \text{bool}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{Ref1:RawRefVal} &:: — \Rightarrow \text{Ref2:RawRefVal} :: — \\
& (\text{Ref1 } =_{K} \text{Ref2}) :: \text{bool}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{TE1:K} &\neq \text{TE2:K} \\
& ! (\text{TE1 } \Rightarrow_{\text{TE2}})
\end{align*}
\]

RULE

\[
\begin{align*}
\text{I1:Int} &:: — < \text{I2:Int} :: — \\
& (\text{I1 } <_{\text{Int}} \text{I2}) :: \text{bool}
\end{align*}
\]
RULE

**F1**:Float :: — < **F2**:Float :: —

\( (\text{F1} \ <_{\text{Float}} \text{F2}) :: \text{bool} \)

RULE

**I1**:Int :: — > **I2**:Int :: —

\( (\text{I1} \ >_{\text{Int}} \text{I2}) :: \text{bool} \)

RULE

**F1**:Float :: — > **F2**:Float :: —

\( (\text{F1} \ >_{\text{Float}} \text{F2}) :: \text{bool} \)

RULE

**I1**:Int :: — <= **I2**:Int :: —

\( (\text{I1} \ <=_{\text{Int}} \text{I2}) :: \text{bool} \)

RULE

**F1**:Float :: — <= **F2**:Float :: —

\( (\text{F1} \ <=_{\text{Float}} \text{F2}) :: \text{bool} \)

RULE

**I1**:Int :: — >= **I2**:Int :: —

\( (\text{I1} \ >=_{\text{Int}} \text{I2}) :: \text{bool} \)

RULE

**F1**:Float :: — >= **F2**:Float :: —

\( (\text{F1} \ >=_{\text{Float}} \text{F2}) :: \text{bool} \)

**A.2.2** Numeric operators

RULE

**I1**:Int :: NT1:Type ≪ **I2**:Int :: —

\( (\text{I1} \ll_{\text{Int}} (\text{I2} \&_{\text{Int}} (\text{bitCount}(\text{normalizeType}(\text{NT1})) \ -_{\text{Int}} 1))) :: \text{tempType} \)

RULE

**I1**:Int :: NT1:Type ≫ **I2**:Int :: —

\( (\text{I1} \gg_{\text{Int}} (\text{I2} \&_{\text{Int}} (\text{bitCount}(\text{normalizeType}(\text{NT1})) \ -_{\text{Int}} 1))) :: \text{tempType} \)
RULE

\[ \begin{align*}
I1 : Int &:: NT1 : Type \Rightarrow I2 : Int :: NT2 : NumericType \\
\quad &(( (I1 \geq_{Int} 0) :: bool ) ? (I1 :: NT1 \searrow I2 :: NT2) \\
\quad &:: ( (I1 :: NT1 \searrow I2 :: NT2) \uparrow (2 :: NT1 \leftarrow (\sim I2 :: NT2)) )
\end{align*} \]

RULE

\[ \begin{align*}
I1 : Int &:: \quad + \quad I2 : Int :: \quad - \\
\quad &((I1 +_{Int} I2) :: tempType)
\end{align*} \]

RULE

\[ \begin{align*}
F1 : Float &:: \quad + \quad F2 : Float :: \quad - \\
\quad &((F1 +_{Float} F2) :: tempType)
\end{align*} \]

RULE

\[ \begin{align*}
I1 : Int &:: \quad - \quad I2 : Int :: \quad - \\
\quad &((I1 -_{Int} I2) :: tempType)
\end{align*} \]

RULE

\[ \begin{align*}
F1 : Float &:: \quad - \quad F2 : Float :: \quad - \\
\quad &((F1 -_{Float} F2) :: tempType)
\end{align*} \]

RULE

\[ \begin{align*}
I1 : Int &:: \quad * \quad I2 : Int :: \quad - \\
\quad &((I1 *_{Int} I2) :: tempType)
\end{align*} \]

RULE

\[ \begin{align*}
F1 : Float &:: \quad * \quad F2 : Float :: \quad - \\
\quad &((F1 *_{Float} F2) :: tempType)
\end{align*} \]

RULE

\[ \begin{align*}
I1 : Int &:: \quad / \quad I2 : Int :: \quad - \\
\quad &((I1 \div_{Int} I2) :: tempType)
\end{align*} \]

REQUIRES \[ I2 =/=_{Int} 0 \]

RULE

\[ \begin{align*}
F1 : Float &:: \quad / \quad F2 : Float :: \quad - \\
\quad &((F1 \div_{Float} F2) :: tempType)
\end{align*} \]
RULE

\[ \text{throw new class String2Id ("java.lang.ArithmeticException")(/ by zero) ;} \]

RULE

\[
\text{I1: Int :: -- } \% \text{ I2: Int :: -- } \\
(I_1 \%_{\text{Int}} I_2) :: \text{tempType} \\
\text{REQUIRES I2 =/= Int 0} \\
\]

RULE

\[
\text{F1: Float :: -- } \% \text{ F2: Float :: -- } \\
(F_1 \%_{\text{Float}} F_2) :: \text{tempType} \\
\]

RULE

\[
\text{throw new class String2Id ("java.lang.ArithmeticException")(/ by zero) ;} \\
\]

RULE NormalizeLeftOperandToFloat

\[
\text{KL: KLabel}(I_{\text{Int}} :: -- : \text{IntType}, -- : \text{Float} :: -- : \text{FloatType}) \\
\text{Int2Float(I)} :: \text{float} \\
\text{REQUIRES isFloatBinaryOp (KL)} \\
\]

RULE NormalizeRightOperandToFloat

\[
\text{KL: KLabel(-- : \text{Float} :: -- : \text{FloatType}, I_{\text{Int}} :: -- : \text{IntType})} \\
\text{Int2Float(I)} :: \text{float} \\
\text{REQUIRES isFloatBinaryOp (KL)} \\
\]

SYNTAX

\[
K_{\text{Item}} ::= \text{isFloatBinaryOp (K_{\text{Label}}) [function]} \\
\]

RULE

\[
\text{isFloatBinaryOp (KL: KLabel)} \\
\text{KL = KLabel 'Plus \lor Boolean KL = KLabel 'Minus \lor Boolean} \\
\text{KL = KLabel 'Mul \lor Boolean KL = KLabel 'Div} \\
\text{\lor Boolean KL = KLabel 'Remain \lor Boolean} \\
\text{KL = KLabel 'Lt \lor Boolean KL = KLabel 'LtEq \lor Boolean} \\
\text{KL = KLabel 'Gt \lor Boolean KL = KLabel 'GtEq \lor Boolean} \\
\text{KL = KLabel 'Eq \lor Boolean KL = KLabel 'NotEq} \\
\]

RULE

\[
( ++ \text{loc (L: Int) :: NT: NumericType}) \\
\text{loc (L) :: NT += rightTypedNumber (1, NT)} \\
\]
RULE

\[
\frac{( \text{ -- loc (L:Int) :: NT:NumericType} )}{\text{loc (L) :: NT} = \text{rightTypedNumber (1, NT)}}
\]

RULE

\[
\begin{align*}
\text{! B:Bool} & : \text{bool} \\
\frac{( \sim_{\text{Bool}} B )}{\text{:: bool}}
\end{align*}
\]

RULE

\[
\begin{align*}
\sim \text{I:Int} & : - \\
\frac{( \sim_{\text{Int}} I )}{\text{:: tempType}}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{+ I:Int} & : - \\
\frac{\text{I} \text{:: tempType}}{}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{+ F:Float} & : - \\
\frac{\text{F} \text{:: tempType}}{}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{+ F:Float} & : - \\
\frac{\text{F} \text{:: tempType}}{}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{\text{loc (L:} & \text{Int) :: NT:NumericType ++} \\
\frac{( \text{++ loc (L) :: NT} )}{\text{-- loc (L) :: NT} = \text{rightTypedNumber (1, NT)}}
\end{align*}
\]

RULE

\[
\begin{align*}
\text{loc (L:} & \text{Int) :: NT:NumericType --} \\
\frac{( \text{-- loc (L) :: NT} )}{\text{+ \text{rightTypedNumber (1, NT)}}}
\end{align*}
\]

Computes a TypedVal where value representation is of correct numeric type: Int when NT is IntType,
Float when NT is FloatType

**SYNTAX**  \[ KItem ::= \texttt{rightTypedNumber} \left( \texttt{Int}, \texttt{NumericType} \right) \] [function]

**RULE**

\[
\begin{align*}
\texttt{rightTypedNumber} \ (I: \texttt{Int}, \texttt{IntT}: \texttt{IntType}) \\
I : : \texttt{IntT}
\end{align*}
\]

**RULE**

\[
\begin{align*}
\texttt{rightTypedNumber} \ (I: \texttt{Int}, \texttt{FloatT}: \texttt{FloatType}) \\
\texttt{Int2Float}(I) : : \texttt{FloatT}
\end{align*}
\]

### A.2.3 String operators

**RULE**

\[
\begin{align*}
\texttt{Str1}: \texttt{String} & : : \texttt{—} + \texttt{Str2}: \texttt{String} : : \texttt{—} \\
\ (\texttt{Str1} +_{\texttt{String}} \texttt{Str2}) & : : \texttt{class \ String}
\end{align*}
\]

**RULE**

\[
\begin{align*}
\texttt{Str}: \texttt{String} & : : \texttt{Class:ClassType} + \frac{\texttt{KR:KResult}}{\texttt{toString(KR)}} \\
\texttt{toString}(\texttt{KR}) & \texttt{requires} (\texttt{(( type0f(KR) \neq K \texttt{class String } \land_{\texttt{Bool}} \texttt{Class = K class String } ) \lor_{\texttt{Bool}} (KR = K (\texttt{null} : : \texttt{class String })))})
\end{align*}
\]

**RULE**

\[
\begin{align*}
\frac{\texttt{KR:KResult}}{\texttt{toString(KR)}} + \texttt{Str}: \texttt{String} & : : \texttt{Class:ClassType} \\
\texttt{toString}(\texttt{KR}) & \texttt{requires} (\texttt{(( type0f(KR) \neq K \texttt{class String } \land_{\texttt{Bool}} \texttt{Class = K class String } ) \lor_{\texttt{Bool}} (KR = K (\texttt{null} : : \texttt{class String })))})
\end{align*}
\]

### A.2.4 Conditional operator

Conditional expression : ?: . Desugared into an if with cast. The biggest difficulty is computing the expression type, according to JLS1 §15.24 The whole expression should always come wrapped into a cast, from elaboration. Thus we don’t have to worry about proper type conversion here.

**RULE**

\[
\begin{align*}
\texttt{CondExp}: \texttt{Exp} \ ? \texttt{TrueExp}: \texttt{Exp} & : \texttt{FalseExp}: \texttt{Exp} \\
\texttt{ifAux} \ (\texttt{CondExp}, \texttt{TrueExp}, \texttt{FalseExp})
\end{align*}
\]

126
A.2.5 Assignment operators

Both basic assignment and compound assignments like +=, -=, etc.

**CONTEXT**

\[-: KResult = \square\]

**SYNTAX**

\[LHS ::= TypedVal\]

**RULE Assign**

\[
\begin{align*}
\text{loc } (L: Int) &:: \text{ Res}\cdot Type = V: RawVal :: \text{ Res}\cdot T \\
\text{store } (L, V :: \text{ Res}\cdot T) &\cup V :: \text{ Res}\cdot T
\end{align*}
\]

Desugars compound assign into the underlying operator + assign.

**RULE**

\[
\begin{align*}
\text{loc } (L: Int) &:: \text{ T}\cdot Type \ast= \text{ Exp2}\cdot K \\
\text{loc } (L) &:: \text{ T} = \text{ cast } (\text{ T}, \text{ lookup } (L, \text{ T}) \ast \text{ Exp2})
\end{align*}
\]

**RULE**

\[
\begin{align*}
\text{loc } (L: Int) &:: \text{ T}\cdot Type \slash= \text{ Exp2}\cdot K \\
\text{loc } (L) &:: \text{ T} = \text{ cast } (\text{ T}, \text{ lookup } (L, \text{ T}) \slash \text{ Exp2})
\end{align*}
\]

**RULE**

\[
\begin{align*}
\text{loc } (L: Int) &:: \text{ T}\cdot Type \equiv= \text{ Exp2}\cdot K \\
\text{loc } (L) &:: \text{ T} = \text{ cast } (\text{ T}, \text{ lookup } (L, \text{ T}) \equiv \text{ Exp2})
\end{align*}
\]

**RULE**

\[
\begin{align*}
\text{loc } (L: Int) &:: \text{ T}\cdot Type \ast= \text{ Exp2}\cdot K \\
\text{loc } (L) &:: \text{ T} = \text{ cast } (\text{ T}, \text{ lookup } (L, \text{ T}) \ast \text{ Exp2})
\end{align*}
\]

**RULE**

\[
\begin{align*}
\text{loc } (L: Int) &:: \text{ T}\cdot Type \equiv= \text{ Exp2}\cdot K \\
\text{loc } (L) &:: \text{ T} = \text{ cast } (\text{ T}, \text{ lookup } (L, \text{ T}) \equiv \text{ Exp2})
\end{align*}
\]

127
A.2.6 Cast operator — primitive types
The type of RV could either be NumericType or tempType. This rule matches both.

**rule cast-Number**
\[
\text{cast (NT:NumericType, RV:RawVal :: —) }
\]
\[
\text{normalize (RV :: NT)}
\]

**rule cast-Bool**
\[
\text{cast (bool, B:Bool :: bool ) }
\]
\[
\text{B :: bool}
\]

Artificial cases of cast, arised during semantics separation

**rule cast-void**
\[
\text{cast (void, RV:RawVal :: void ) }
\]
\[
\text{RV :: void}
\]
RULE CAST-NOVALUE

\[
\text{cast } (\neg, \ 'K') \\

\]

\[\ 'K\]

A.2.7 Heating/cooling rules for lvalue

RULE

\[
\begin{array}{c}
\ 'K' \\
\text{lvalue} (K) \\
\end{array} \succ \left( \begin{array}{c}
\text{\small{K:K}} \\
\text{=} \\
\Box \\
\end{array} \right)
\]

REQUIRES \neg \text{Bool} (\text{isKResult(K)} =_K \text{true})

RULE

\[
\begin{array}{c}
\ 'K' \\
\text{lvalue} (K) \\
\end{array} \succ \left( \begin{array}{c}
\text{\small{K:K}} \\
\text{\ast=} \\
\Box \\
\end{array} \right)
\]

REQUIRES \neg \text{Bool} (\text{isKResult(K)} =_K \text{true})

RULE

\[
\begin{array}{c}
\ 'K' \\
\text{lvalue} (K) \\
\end{array} \succ \left( \begin{array}{c}
\text{\small{K:K}} \\
\text{/=} \\
\Box \\
\end{array} \right)
\]

REQUIRES \neg \text{Bool} (\text{isKResult(K)} =_K \text{true})

RULE

\[
\begin{array}{c}
\ 'K' \\
\text{lvalue} (K) \\
\end{array} \succ \left( \begin{array}{c}
\text{\small{K:K}} \\
\text{%=} \\
\Box \\
\end{array} \right)
\]

REQUIRES \neg \text{Bool} (\text{isKResult(K)} =_K \text{true})

RULE

\[
\begin{array}{c}
\ 'K' \\
\text{lvalue} (K) \\
\end{array} \succ \left( \begin{array}{c}
\text{\small{K:K}} \\
\text{+=} \\
\Box \\
\end{array} \right)
\]

REQUIRES \neg \text{Bool} (\text{isKResult(K)} =_K \text{true})

RULE

\[
\begin{array}{c}
\ 'K' \\
\text{lvalue} (K) \\
\end{array} \succ \left( \begin{array}{c}
\text{\small{K:K}} \\
\text{-=} \\
\Box \\
\end{array} \right)
\]

REQUIRES \neg \text{Bool} (\text{isKResult(K)} =_K \text{true})
RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
\triangleright
\hline
\mathbf{K} : \mathbf{K} \\
\ni
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
\triangleright= \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
\triangleright>= \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
\&= \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
\hat{=} \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
\|= \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
++ \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

RULE

\[
\begin{array}{c}
\kappa \\
\hline
\text{lvalue}(\mathbf{K})
\end{array}
\quad \begin{array}{c}
- \\
\hline
\mathbf{K} : \mathbf{K} \\
\i
\end{array}
\]

REQUIRES \neg \text{Bool} \ (\text{isKResult}(\mathbf{K}) = \mathbf{K} \text{ true})

130
RULE

\[
\begin{array}{c}
\kappa \\
\text{lvalue}(\kappa)
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\kappa \\
\text{K}
\end{array} \quad ++ \\
\text{requires} \quad \neg \text{Bool} \left( \text{isKResult}(\kappa) = \kappa \text{ true} \right)
\]

RULE

\[
\begin{array}{c}
\kappa \\
\text{lvalue}(\kappa)
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\kappa \\
\text{K}
\end{array} \quad -- \\
\text{requires} \quad \neg \text{Bool} \left( \text{isKResult}(\kappa) = \kappa \text{ true} \right)
\]

RULE

\[
\begin{array}{c}
\text{KResult} \quad \kappa \\
\text{KResult}
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\boxempty
\end{array} = \quad \text{KResult}
\]

RULE

\[
\begin{array}{c}
\text{KResult} \quad \kappa \\
\text{KResult}
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\boxempty
\end{array} * = \quad \text{KResult}
\]

RULE

\[
\begin{array}{c}
\text{KResult} \quad \kappa \\
\text{KResult}
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\boxempty
\end{array} / = \quad \text{KResult}
\]

RULE

\[
\begin{array}{c}
\text{KResult} \quad \kappa \\
\text{KResult}
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\boxempty
\end{array} \% = \quad \text{KResult}
\]

RULE

\[
\begin{array}{c}
\text{KResult} \quad \kappa \\
\text{KResult}
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\boxempty
\end{array} += \quad \text{KResult}
\]

RULE

\[
\begin{array}{c}
\text{KResult} \quad \kappa \\
\text{KResult}
\end{array} \quad \rightsquigarrow \quad \begin{array}{c}
\boxempty
\end{array} -= \quad \text{KResult}
\]
RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \iff \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \gg \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \gg\gg \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \& \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \hat{=} \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \mid \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \to \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad (\square \to \bot) \quad KR
\]

RULE
\[
\frac{KR:KResult \models \kappa}{\Downarrow} \quad \square \to \bot \quad KR
\]
Required when a JVM-related exception (e.g. produced by the semantics at runtime) is thrown inside a lvalue.

**Rule lvalue-throw-desugar**

\[
\text{lvalue ( throw } \text{E:Exp ; )}
\]

\[
\text{throw E ;}
\]

[structural]

### A.3 Module EXPRESSIONS-CLASSES

Expressions that operate over objects.

#### A.3.1 Instanceof operator

**Rule InstanceOf**

\[
\text{V:RawVal :: instanceof RT2:RefType}
\]

\[
( \text{V } \neq \text{null }) \&\& \text{subtype ( typeof (V), RT2 )}
\]

#### A.3.2 Cast operator — reference types

**Rule cast-RefType**

\[
\text{cast (RT1:RefType, V:RawVal :: RT2:RefType)}
\]

\[
\text{ifAux ( subtype ( typeof (V), RT1), V :: RT1, throw new class String2Id (}
\]

"java.lang.ClassCastException"( toExps ( [ ( toString ( typeof (V))

+ " cannot be cast to " ) + toString (RT1 ) ] ) ) ;

#### A.3.3 Auxiliary constructs produced during elaboration

**Rule**

\[
\text{stmtAndExp ( Stmt:K, Exp:K)}
\]

\[
\text{Stmt } \triangleleft \text{ Exp}
\]
A.4 Module STATEMENTS

A.4.1 Call of the main method

\[
\text{rule ExecutionPhase-Start}
\]
\[
\begin{align*}
\text{getClassType} & \left( \text{packageName} ( \text{StringId} ("")), \text{StringId} (\text{MainClassS}) \right) \cdot \text{StringId} ("main") (\text{toExps} (\left[ \text{\'NewArray} (\text{class String}, [\text{\'Dim} (0 :: \text{int})], [\text{\'KList} ] ) ] ) ) ) ;
\end{align*}
\]

\[
\langle \text{Map} \rangle \text{env} \quad \langle \text{MainClassS}:\text{String} \rangle \text{mainClass} \quad \langle \text{UnfoldingPhase} \rangle \text{ExecutionPhase} \quad \langle \text{globalPhase} \rangle
\]

A.4.2 Blocks

JLS §14.2

\[
\text{rule Block}
\]
\[
\begin{align*}
\{ S : K \} & \quad \rightarrow_k \quad \langle \text{env} \rangle \\
\langle \text{env} \rangle & \quad \text{Restore the env cell content from env(...).}
\end{align*}
\]

SYNTAX \[\text{KItem ::= env} (\text{Map})\]

\[
\text{rule env}
\]
\[
\begin{align*}
\text{env} (\text{Env:Map}) & \quad \rightarrow_k \quad \langle \text{Env} \rangle \\
\langle \text{Env} \rangle & \quad \text{[structural]}
\end{align*}
\]

\[
\text{rule env-double-Discard}
\]
\[
\begin{align*}
\text{env} (\text{---}) & \quad \rightarrow_k \quad \langle \text{Env} \rangle \\
\langle \text{Env} \rangle & \quad \text{[structural]}
\end{align*}
\]

A.4.3 Local Variable Declarations

JLS §14.4 Not only local but fields also
A.4.4 Empty statement
JLS §14.6
\textbf{RULE EMPTY}

\begin{align*}
\vdash & \quad \kappa
\end{align*}

A.4.5 Labeled statements
JLS §14.7
\textbf{RULE LABELED}

\begin{align*}
X\cdot Id : S\cdot Stmt & \\
S & \leadsto \text{labeledImpl} (X)
\end{align*}

Processed version of the labeled statement

\textbf{SYNTAX} \quad K\text{Item} ::= \text{labeledImpl} (Id)

\textbf{RULE LABELEDIMPL-DISCARD}

\begin{align*}
labeledImpl (---) & \\
\quad \kappa
\end{align*}

A.4.6 Expression statements
JLS §14.8
\textbf{RULE}

\begin{align*}
--- : \text{TypedVal} & \\
\quad \kappa
\end{align*}
A.4.7 If statement

**RULE If-Then-Desugar**

\[
\text{if (E:K) } S:\text{Stmt} \\
\text{if (E) } S \text{ else ( ; )}
\]

**RULE If-True**

\[
\text{if (true :: bool) } S:\text{Stmt} \text{ else } --:\text{Stmt} \\
S \\
\text{[symbolic-rule]}
\]

**RULE If-False**

\[
\text{if (false :: bool) } --:\text{Stmt} \text{ else } S:\text{Stmt} \\
S \\
\text{[symbolic-rule]}
\]

A.4.8 Assert statement

**RULE AssertStm-True-OneArg**

\[
\text{assert true :: bool ; } \\
\text{`K}
\]

**RULE AssertStm-False-OneArg**

\[
\text{assert false :: bool ; } \\
\text{throw new class String2Id ("java.lang.AssertionError")( null :: class String ) ; }
\]

**RULE AssertStm-FirstTrue-SecondDiscarded**

\[
\text{assert true :: bool : -- ; } \\
\text{`K}
\]

**CONTEXT**

\[
\text{assert false :: bool : □ ;}
\]

**RULE AssertStm-FirstFalse**

\[
\text{assert (false :: bool) : (TV:TypedVal) ; } \\
\text{throw new class String2Id ("java.lang.AssertionError")( toString(TV)) ;}
\]
A.4.9 Switch statement

**SYNTAX**
\[ K\text{Item} ::= \text{switchImpl( TypedVal, KListWrap, K) } \]

The switchEnd in the right-hand side of => is required to properly interact with break.

**rule Switch**
\[
\text{'Switch}(TV:\text{TypedVal}, \text{'SwitchBlock([ Ks:KList ], TrailingLabels:K)}) \]
\[ \sim \text{switchImpl} (TV, [ Ks, \text{'SwitchGroup}(\text{TrailingLabels, 'K }), 'K ]) \sim \text{switchEnd} \]

**CONTEXT**
\[ \text{switchImpl} (---, [ \text{'SwitchGroup([ 'Case(□), ---:KList ], ---), ---:KList ], ---) } \]

The type associated to V and V2 is not important for match, only the value. JLS3 §14.11, page 377: Every case constant expression associated with a switch statement must be assignable ($5.2$) to the type of the switch Expression.

**rule switchImpl-CaseNotMatch**
\[ \text{switchImpl} (V:\text{RawVal} :: ---, [ \text{'SwitchGroup([ 'Case(V2:RawVal :: ---) \sim \text{KList}, ---), ---:KList ], ---) } \]
\[ \text{---}, ---) \]

REQUIRES \( V \neq \_ \_ V2 \)

Once case of first switch group matches, the whole switch is replaced by it’s block. The execution rules for switch will discard switch-related constructs and will execute the statements.

**rule switchImpl-CaseMatch**
\[ \text{switchImpl} (V:\text{RawVal} :: ---, [ \text{'SwitchGroup([ 'Case(V :: ---), ---:KList ], S:K), Ks:KList ], ---) } \]
\[ \text{---} \]
\[ \text{[ S, Ks ]} \]

**rule switchImpl-DefaultSave**
\[ \text{switchImpl} (TV:\text{TypedVal}, [ \text{'SwitchGroup([ 'Default(---) \sim \text{KList}, ---:KList ], S:K), Ks:KList ], \]
\[ ---:K \sim \text{KList}) \]
\[ \text{[ S, Ks ]} \]

**rule switchImpl-SwitchGroup-Discard**
\[ \text{switchImpl} (TV:\text{TypedVal}, [ \text{'SwitchGroup([ 'KList ], ---) } \]
\[ \text{---}, ---) \]

137
rule switchImpl-Default-Exec

\[
\text{switchImpl}(\_]K\text{List}\_, \text{DefaultStm}:K) \\
\text{DefaultStm}
\]

rule SwitchGroup-Exec

\[
\text{SwitchGroup}(\_]K\_, \text{S}:K) \\
\text{S}
\]

SYNTAX \[KItem ::= \text{switchEnd}\]

rule switchEnd-Discard

\[
\text{switchEnd} \\
\_]K
\]

A.4.10 While statement
and loop infrastructure.

rule while

\[
\text{while}(E:\text{Exp})S:\text{Stmt} \\
\text{whileImpl}(E, S) \\
[\text{structural}]
\]

SYNTAX \[Stmt ::= \text{whileImpl}(K, K)\]

rule whileImpl

\[
\text{whileImpl}(E,K,S,K) \\
\text{if}(E)\{S \sim \text{whileImpl}(E, S)\} \\
[\text{structural}]
\]

A.4.11 Do statement

rule do-while

\[
do S:\text{Stmt} \text{while}(E:\text{Exp}) ; \\
S \sim \text{whileImpl}(E, S) \\
[\text{structural}]
\]
A.4.12 For statement

**SYNTAX** \( Stmt ::= stm \ ( KListWrap ) \)

**RULE stm-KListWrap-empty**
\[
stm \ ( [ 'KList ] ) \quad \overset{'K}{\text{structural}}
\]

**RULE stm-KListWrap-exp**
\[
stm \ ( [ E:Exp, Es:KList ] ) \quad \overset{E; \sim \ stm \ ( [ Es ] )}{\text{structural}}
\]

**RULE getKLabel ( E ) =_{KLabel} 'cast**

**RULE stm-KListWrap-LocalVarDec**
\[
stm \ ( [ K:K, Ks:KList ] ) \quad \overset{[ K, Ks ]}{\text{structural}}
\]

**RULE For-FirstArgV1Red**
\[
\overset{'For\ ( [ InitExps:KList ] ), \sim :KList)}{\text{structural}}
\]

**RULE For-FirstArgV2Red**
\[
\overset{'LocalVarDec(Ks:KList), \sim :KList)}{\text{structural}}
\]

**RULE For-SecondArgRed**
\[
\overset{'One('KList'), \sim :KList)}{\text{structural}}
\]

**RULE For-ThirdArgRed**
\[
\overset{'Some(true), \sim :KList)}{\text{structural}}
\]
rule For
\[
\text{′For(stm (InitClause:KListWrap), ′Some(TestExp:K), stm (UpdClause:KListWrap), S:K)}
\]
\[
\text{′Block(stm (InitClause) \sim ifAux (TestExp, (′Block(S) \sim whileImpl (stm (UpdClause) \sim TestExp, S)), ′K))}
\]

[structural]

A.4.13 Break statement

rule Break-UnlabeledPropagate
\[
\text{break ′None(—) ; } \overset{\text{KL:KLabel(—) }}{\sim} \overset{′K}{\downarrow}
\]
\[
\text{requires KL \neq_{KLabel} ′env \wedge_{Bool} KL \neq_{KLabel} ′finallyBlock \wedge_{Bool} KL \neq_{KLabel} ′whileImpl \wedge_{Bool} KL}
\]
\[
\neq_{KLabel} ′switchEnd
\]

rule Break-Unlabeled
\[
\text{break ′None(—) ; } \overset{\text{KL:KLabel(—) }}{\sim} \overset{′K}{\downarrow}
\]
\[
\text{requires KL =_{KLabel} ′whileImpl \vee_{Bool} KL =_{KLabel} ′switchEnd}
\]

rule Break-LabeledPropagate
\[
\text{break ′Some(X:Id) ; } \overset{\text{KL:KLabel(—) }}{\sim} \overset{′K}{\downarrow}
\]
\[
\text{requires KL \neq_{KLabel} ′env \wedge_{Bool} KL \neq_{KLabel} ′finallyBlock \wedge_{Bool} KL \neq_{KLabel} ′labeledImpl}
\]

rule BreakLabeledNotMatch
\[
\text{break ′Some(X1:Id) ; } \overset{\text{labeledImpl(X2:Id) }}{\sim} \overset{′K}{\downarrow}
\]
\[
\text{requires X1 \neq_{K} X2}
\]

rule Break-LabeledMatch
\[
\text{break ′Some(X:Id) ; } \overset{\text{labeledImpl(X:Id) }}{\sim} \overset{′K}{\downarrow}
\]
A.4.14 Continue statement

rule Continue-Propagate

\[
\text{continue } \leadsto \text{KL} \in \text{KLabel} \quad \text{requires } \text{KL} \neq \text{KLabel}' \text{env} \land \text{Bool} \quad \text{KL} \neq \text{KLabel}' \text{finallyBlock} \land \text{Bool} \quad \text{KL} \neq \text{KLabel}' \text{whileImpl}
\]

rule Continue-Unlabeled

\[
\text{continue }'\text{None} \quad \leadsto \text{whileImpl} \quad \text{requires } \text{KL} \neq \text{KLabel}' \text{env} \land \text{Bool} \quad \text{KL} \neq \text{KLabel}' \text{finallyBlock} \land \text{Bool} \quad \text{KL} \neq \text{KLabel}' \text{whileImpl}
\]

rule Continue-LabeledMeetsWhile

\[
\text{whileLabel} \left( \text{LabelK} : \text{K} \right) \quad \leadsto \text{continue }'\text{Some}(\text{X} : \text{Id}) \quad \text{requires } \text{LabelK} \neq \text{K} \text{X}
\]

rule Continue-LabeledNotMatch

\[
\text{whileLabel} \quad \leadsto \text{continue }'\text{Some}(\text{X} : \text{Id}) \quad \text{requires } \text{LabelK} \neq \text{K} \text{X}
\]

rule Continue-LabeledMatch

\[
\text{whileLabel} \quad \leadsto \text{continue }'\text{Some}(\text{X} : \text{Id}) \quad \text{requires } \text{LabelK} \neq \text{K} \text{X}
\]

Auxiliary constructs used in the semantics of continue. Search for the label associated with the last encountered while statement

SYNTAX

\[
\text{Kitem} ::= \text{getWhileLabel} \left( \text{K} \right) \quad \left| \quad \text{whileLabel} \left( \text{K} \right) \right.
\]

rule

\[
\text{getWhileLabel} \quad \text{requires } \text{env} \quad \leadsto \text{whileLabel} \left( \text{env} \right)
\]

rule

\[
\text{getWhileLabel} \left( \text{K} \right) \quad \text{whileLabel} \left( \text{K} \right)
\]
RULE
\[
\text{getWhileLabel} ( \text{labeledImpl} (X:Id) \xrightarrow{\kappa} \cdot ) \quad \text{whileLabel} (X)
\]

RULE
\[
\text{getWhileLabel} (KL:KLabel(\cdot ) \xrightarrow{\kappa} \cdot ) \\
\text{whileLabel} (\cdot K)
\]

REQUIRES KL \neq KLabel 'env \land Bool KL \neq KLabel 'labeledImpl

A.4.15 Return statement

CONTEXT
\text{return 'Some(\emptyset ) ;}

RULE RETURN-NONE-DESGAR

\text{return 'None(\cdot ) ;}
\begin{align*}
\text{return 'Some(none :: void) ;} \\
\text{[structural]}
\end{align*}

RULE RETURN-PROPAGATE

\text{return 'Some(TV:TypedVal) ;} \xrightarrow{KL:KLabel(Ks:KList)} ^\kappa

REQUIRES KL \neq KLabel 'env \land Bool KL \neq KLabel 'finallyBlock

RULE RETURN-METHODEND

\begin{align*}
\langle \text{return 'Some(TV:TypedVal) ;} \rangle & \xrightarrow{k} (TV \xrightarrow{\kappa} K) \\
\langle \text{sl (K:K, MethContext:Bag)} \rangle & \xrightarrow{\text{List}} \text{stack} \\
\langle \text{-:Bag} \rangle & \xrightarrow{\text{methodContext}} \text{methodContext}
\end{align*}

A.4.16 Throw statement

Exceptions are propagated until a catch compatible with them is encountered.

RULE THROW
\[
\xrightarrow{\kappa} \text{throw TV:TypedVal ;} \xrightarrow{\text{catchBlocks (}} \\
\text{checkCatch ( subtype ( typeOf (TV), T))} \\
\text{catchImpl ('ParamImpl (T:Type, X:Id), \cdot ) \xrightarrow{-:CatchClauses})}
\]
**SYNTAX**  \( KItem ::= \text{checkCatch}(K) \) [strict]

**RULE THROW-checkCatch-True**
\[
\begin{align*}
\text{checkCatch}(\text{true} :: \text{bool}) & \overset{\text{strict}}{\leadsto} \text{throw TV}:\text{TypedVal} ; \text{catchBlocks}(\text{catchImpl}(\text{Param} : \text{Param}, \text{CatchS} : K) \rightarrow \text{CatchClauses}) \\
\{ \text{initParams}(\text{Param}, \text{TV}) \rightarrow \text{CatchS} \}
\end{align*}
\]

**RULE THROW-checkCatch-False**
\[
\begin{align*}
\text{checkCatch}(\text{false} :: \text{bool}) & \overset{\text{strict}}{\leadsto} \text{throw TV}:\text{TypedVal} ; \text{catchBlocks}(\text{catchImpl}(\_, \_) \rightarrow \text{CatchClauses}) \\
\text{C}
\end{align*}
\]

**RULE THROW-catchBlocks-EmptyDiscard**
\[
\begin{align*}
\text{throw TV}:\text{TypedVal} ; \text{catchBlocks}(\rightarrow \text{CatchClauses}) \overset{\text{strict}}{\leadsto} \text{K}
\end{align*}
\]

**RULE THROW-Propagate**
\[
\begin{align*}
\text{throw TV}:\text{TypedVal} ; \text{KL}:\text{KLabel}(\_ \rightarrow \text{K})
\end{align*}
\]

**REQUIRES** \( \text{KL} \neq \text{KLabel} \rightarrow \text{env} \land \text{Bool} \) \( \text{KL} \neq \text{KLabel} \rightarrow \text{finallyBlock} \land \text{Bool} \) \( \text{KL} \neq \text{KLabel} \rightarrow \text{catchBlocks} \)

**RULE THROW-MethodEnd**
\[
\begin{align*}
\langle \text{throw TV}:\text{TypedVal} ; \rightarrow \text{K} \rangle \overset{\text{stack}}{\leadsto} \text{sl}(\text{K} : \text{K}, \text{MethContext} : \text{Bag}) \rightarrow \text{List} \rightarrow \text{methodContext} \rightarrow \text{MethContext}
\end{align*}
\]

**RULE THROW-CausesThreadTermination**
\[
\begin{align*}
\langle \text{Field(class String2Id(\"java.lang.System\"), String2Id("out")}. String2Id("println") \rightarrow \text{cast( class Object, \"Plus\"("Thread terminated with exception: ", TV\")))) ; \rightarrow \text{List} \rangle \rightarrow \text{stack}
\end{align*}
\]
A.4.17  Try statement

**RULE Try-Catch-Finally-Desugar**

\[
\text{try } \text{TryS:K} (\text{K:\text{CatchClause} Ks:\text{CatchClauses}}) \text{ finally } \text{FinallyS:K}
\]

\[
\text{try } \{ \text{try } \text{TryS( K Ks) } \} ^{\text{\text{CatchClauses}}} \text{ finally } \text{FinallyS}
\]

**structural**

**RULE Try-Catch**

\[
\text{try } \text{TryS:K} \text{KRs:CatchClauses}
\]

\[
\text{TryS} \rightsquigarrow \text{catchBlocks (KRs)}
\]

**requires** \(\text{isKResult(KRs)}\)

The auxiliary construct that holds catch clauses after try clause has been moved to the top of computation

**SYNTAX** \(KItem ::= \text{catchBlocks ( CatchClauses )}\)

If try clause executed without raising an exception, this construct will reach the top of computation. In this case it should be discarded.

**RULE catchBlocks-Discard**

\[
\text{catchBlocks (—)} \rightsquigarrow \text{K}
\]

**RULE Try-Finally**

\[
\text{try } \text{TryS:K} ^{\text{\text{CatchClauses}}} \text{ finally } \text{FinallyS:K}
\]

\[
\text{TryS} \rightsquigarrow \text{finallyBlock (FinallyS)}
\]

The auxiliary construct that holds finally clause after try clause has been moved to the top of computation

**SYNTAX** \(KItem ::= \text{finallyBlock ( K )}\)

**RULE finallyBlock-NormalEnd**

\[
\text{finallyBlock ( FinallyS:K)}
\]

\[
\text{FinallyS}
\]

Only a throw with its value computed is stack consumer, not any throw. If we’ll define any throw to be StackConsumerStmt, then we’ll get rare problems with ExceptionInInitializerError. Other values for StackConsumerStmt are defined in the syntax.

**RULE**

\[
\text{isStackConsumerStmt(throw —:TypedVal ;)}
\]

\[
\text{true}
\]
A.5 Module CORE-EXEC

A collection of utility functions that many other, unrelated modules depend on. Grouped here to minimize the dependencies between other modules.

ListItem content as a stack layer

SYNTAX  $Kitem ::= sl (K, Bag)$

Auxiliary constructs used in variable access semantics.

SYNTAX  $Kitem ::= lvalue (K)$

SYNTAX  $RawVal ::= loc (Int)$

Store to the given location in <store> (first argument) the given value (second argument).

SYNTAX  $Kitem ::= store (Int, K) [strict(2)]$

Restore the content of the cell <methodContext>

SYNTAX  $Kitem ::= restoreMethContext (Bag)$

RULE restoreMethContext

$\langle \underbrace{restoreMethContext \ (MethContext:Bag)}_{\text{"k"}} \rangle \ k \ \underbrace{\text{methodContext}}_{\text{\overbrace{MethContext}}}$

A.6 Module VAR-LOOKUP

A.6.1 Local variable access

At execution phase $'ExprName(X)$ always represents a local variable.
RULE ExprName-local
\[
\langle '\text{ExprName}(X:id) \rangle_k \xrightarrow{\text{typedLookup}} \langle L \rangle \xrightarrow{\text{env}} X \mapsto L.:\text{Int}
\]

RULE lvalue-ExprName-local
\[
\langle \text{lvalue}(\langle '\text{ExprName}(X:id) \rangle) \rangle_k \xrightarrow{\text{typedLoc}} \langle X \mapsto L.:\text{Int} \rangle_{\text{env}}
\]

A.6.2 Qualified this — self reference

RULE QThis
\[
\langle 'QThis(Class:ClassType) \rangle_k \xrightarrow{\text{lookupQThis}} \langle \text{Class, RV :: CrntClass} \rangle \xrightarrow{\text{CrntClass}} \langle \text{OL.:Int} \rangle_{\text{location}} \xrightarrow{\text{store}} \langle \neg \text{OL} \mapsto RV.:\text{RawVal} :: \neg \rangle_{\text{store}}
\]

Search for the right value representing QThis(Class) — an expression of type Class.this

SYNTAX \( KItem ::= \text{lookupQThis}(\ ClassType, \ TypedVal) \)

RULE lookupQThis-found
\[
\text{lookupQThis}(\langle \text{Class:ClassType}, \text{RV:RawVal} :: \text{Class} \rangle)
\]

When we have QThis target class Class, and we look for it in a target object with a different compile-time type ObjClass, we continue our search in the enclosing object of the layer corresponding to ObjClass. This way we may have O.B < O.A, with layers O.B and O.A having different enclosing instance of O, and we will be able to pick the correct enclosing object inside both O.A and O.B.

RULE lookupQThis-next
\[
\text{lookupQThis}(\langle \text{Class:ClassType}, \text{objectRef} (\langle \text{OId}.:\text{Int} :: \rangle) :: \text{RefClass:ClassType} \rangle)
\]

\text{lookupQThis}(\langle \text{Class:ClassType}, \text{EnclosingObj} \rangle)
\]

REQUIRES Class \( \neq \) RefClass

A.6.3 Instance field access

Has the following form: 'ExprName(Qual, X:Id).

CONTEXT
'Field(\emptyset, \neg)
CONTEXT
lvalue ('Field(\[], \_))

RULE Field-instance

\[
\begin{align*}
\quad & \text{lvalue ('Field(objectRef (OId: Int, \_ ) :: TargetClass: ClassType, X:Id))} \\
\quad & \text{typedLookup (L)} \\
\quad & \text{layerClass \{ \_ \rightarrow L: Int \} \_layerEnv}
\end{align*}
\]

RULE lvalue-field-instance

\[
\begin{align*}
\quad & \text{lvalue ('Field(objectRef (OId: Int, \_ ) :: TargetClass: ClassType, X:Id))} \\
\quad & \text{layerClass \{ \_ \rightarrow L: Int \} \_layerEnv}
\end{align*}
\]

RULE Field-instance-OfNull

\[
\begin{align*}
\quad & \text{Field(null :: \_, \_)} \\
\quad & \text{throw new class NullPointerException ( null :: class String );}
\end{align*}
\]

A.6.4 Static field access

Has the following form: 'ExprName(Class, X:Id).

RULE Field-static

\[
\begin{align*}
\quad & \text{'Field(TargetClass: ClassType, X:Id)} \\
\quad & \text{staticInit (TargetClass) \_ staticFieldLookup (TargetClass, X)}
\end{align*}
\]

RULE lvalue-field-static

\[
\begin{align*}
\quad & \text{lvalue ('Field(TargetClass: ClassType, X:Id))} \\
\quad & \text{staticInit (TargetClass) \_ lvalue (staticFieldLookup (TargetClass, X))}
\end{align*}
\]

The actual implementation of static field lookup. The distinction between this function and 'Field(Class, X) term is that at the moment when this function is called the target clas is surely initialized.

SYNTAX KItem ::= staticFieldLookup ( ClassType, Id )

RULE StaticFieldLookup

\[
\begin{align*}
\quad & \text{staticFieldLookup (TargetClass: ClassType, X:Id)} \\
\quad & \text{typedLookup (L)} \\
\quad & \text{layerClass \{ \_ \rightarrow L: Int \} \_staticEnv}
\end{align*}
\]
A.6.5 Functions for accessing the store

Typed version of lookup and loc

**Syntax**  \( KItem ::= \text{typedLookup} (\text{Int}) \)

**Rule typedLookup**

\[
\begin{align*}
\text{lvalue ( typedLookup (L:Int))} \\
\quad \text{lookup (L, T)} \quad \rightarrow \quad \cdots \quad \text{store}
\end{align*}
\]

**Rule lvalue-typedLookup**

\[
\begin{align*}
lvalue ( \text{typedLookup (L:Int)}) \\
\quad \text{typedLoc (L)}
\end{align*}
\]

**Syntax**  \( KItem ::= \text{typedLoc} (\text{Int}) \)

**Rule typedLoc**

\[
\begin{align*}
\text{typedLoc (L:Int)} \\
\quad \text{loc (L) :\ T} \quad \rightarrow \quad \cdots \quad \text{store}
\end{align*}
\]

**Rule lvalue-typedLoc**

\[
\begin{align*}
lvalue ( \text{typedLoc (L:Int)}) \\
\quad \text{typedLoc (L)}
\end{align*}
\]

Retrieve a value from the store based on its location and the given compile-time type. May be wrapped inside lvalue. If lookup is unwrapped, it evaluates to TypedVal — the store value, of the type T — the second lookup argument. If lookup is wrapped into lvalue, it evaluates to \( \text{loc(0L):T} \).

**Syntax**  \( Exp ::= \text{lookup} (\text{Int, Type}) \)
RULE lookup-Location

\[
\text{lookup} \left( L : \text{Int}, \ T1 : \text{Type} \right) \quad \Rightarrow \quad \text{fieldAccessCheckpoint} \left( L \right) \, \sim \, \text{subtype} \left( T2, \ T1 \right) \, \sim \, \text{true} \, \sim \left( V : T1 \right)
\]

\[
\left\langle \, - L \mapsto V : \text{RawVal :: T2 : Type} \, \rightarrow \right\rangle \text{store}
\]

RULE

\[\text{lvalue} \left( \text{lookup} \left( L : \text{Int}, \ T : \text{Type} \right) \right) \quad \Rightarrow \quad \text{loc} \left( L \right) :: T \quad \text{[structural]}\]

SYNTAX \[K\text{Item} ::= \text{fieldAccessCheckpoint} \left( \text{Int} \right)\]

RULE fieldAccessCheckpoint-Local

\[
\text{fieldAccessCheckpoint} \left( L : \text{Int} \right) \, \rightarrow \, L \rightarrow \text{Local} \, \rightarrow \, \text{storeMetadata}
\]

RULE fieldAccessCheckpoint-Field

\[
\text{fieldAccessCheckpoint} \left( L : \text{Int} \right) \, \rightarrow \, L \rightarrow \text{Field} \, \rightarrow \, \text{storeMetadata}
\]

[transition-threading]

Synchronization checkpoint have to be before the actual assignment, like for lookup.

RULE STORE

\[\text{store} \left( L : \text{Int}, \ V : \text{RawVal :: T : Type} \right) \quad \Rightarrow \quad \text{fieldAccessCheckpoint} \left( L \right) \, \sim \, \text{storeImpl} \left( L, \ V :: \right)\]

SYNTAX \[K\text{Item} ::= \text{storeImpl} \left( \text{Int}, \ K \right)\]

RULE storeImpl

\[
\text{storeImpl} \left( L : \text{Int}, \ V : \text{RawVal :: T : Type} \right) \, \rightarrow \, L \rightarrow \, \text{store} \left( V :: \right)
\]

A.7 Module ARRAYS

A.7.1 Array access

The value in store that corresponds to an array element. The constructor "elem" Type is meant to distinguish between regular memory locations and array elements. Array elements need to be represented separately in
order to support array polymorphism.

**SYNTAX**

\[
KItem ::= RawVal :: elem Type
\]

**CONTEXT**

\[
lvalue (\text{"ArrayAccess}(\Box, \Box))
\]

**CONTEXT**

\[
lvalue (\text{"ArrayAccess}(\Box : KResult, \Box))
\]

**RULE ArrayAccess-to-lookup**

\[
\begin{array}{l}
\text{ArrayAccess} (\Box, \text{L:Int, M:Int}) :: \text{arrayOf T:Type} [\text{N:Int :: } \Box] \\
\text{lookup} (\text{L +Int N, T})
\end{array}
\]

**REQUIRES**

\[
\begin{array}{l}
\text{N} \geq \text{Int 0} \land \text{Bool} (\text{N < Int M})
\end{array}
\]

[structural, anywhere]

**RULE lookup-array-location**

\[
\begin{array}{l}
\text{lookup} (\text{L:Int, T1:Type})
\end{array}
\]

\[
\text{fieldAccessCheckpoint} (\text{L}) \leadsto \text{subtype} (\text{T2, T1}) \leadsto \text{true?} \leadsto (V :: T1)
\]

\[
\text{L} \mapsto V:RawVal :: \text{elem T2:Type } \overset{\text{store}}{\leadsto} \Box
\]

**RULE ArrayAccess-to-ArrayIndexOutOfBoundsException**

\[
\begin{array}{l}
\text{ArrayAccess} (\Box, \text{L:Int, M:Int}) :: \Box [\text{N:Int :: } \Box]
\end{array}
\]

\[
\text{throw new class String2Id ("java.lang.ArrayIndexOutOfBoundsException") (Int2String (N)) ;}
\]

**REQUIRES**

\[
\begin{array}{l}
\text{Bool} ((\text{N} \geq \text{Int 0} \land \text{Bool} (\text{N < Int M}))
\end{array}
\]

[anywhere]

**RULE ArrayAccess-null-to-NullPointerException**

\[
\begin{array}{l}
\text{null :: } \Box [\Box]
\end{array}
\]

\[
\text{throw new class NullPointerException (null :: class String );}
\]

[anywhere]

Array length, as defined in JDK

**RULE Field-array-length**

\[
\text{Field} (\text{arrayRef} (\Box, \text{L:Int, M:Int}) :: \Box, \text{X:Id})
\]

\[
\text{N} :: \text{int}
\]

**REQUIRES**

\[
\text{Id2String (X) == String "length"}
\]

150
A.7.2 Assignment to array elements

RULE STOREIMPL-ARRAY-ELEM
\[
\begin{align*}
\text{storeImpl} & \left( L : \text{Int}, V : \text{RawVal} :: T : \text{Type} \right) \\
\to & \left\langle \begin{array}{c}
K \\
- L \mapsto -
\end{array} \right\rangle
\end{align*}
\]

RULE STOREIMPL-ARRAY-LOCATION-CHECK-TYPE
\[
\begin{align*}
\text{storeImpl} & \left( L : \text{Int}, V : \text{RawVal} :: T : \text{Type} \right) \\
\to & \left\langle \begin{array}{c}
K \\
- L \mapsto -
\end{array} \right\rangle
\end{align*}
\]

\[
\begin{align*}
\begin{multlined}
\text{ifAux} \left( \text{subtype} \left( \text{typeof} \left( V \right), \text{StoreType} \right), \text{storeImpl} \left( L, V :: \text{StoreType} \right), \\
\text{throw new class String2Id \left( "java.lang.ArrayStoreException" \right) \left( V :: T . \left( \left( \text{String2Id \left( "getClass" \right) \left( \left( \text{String2Id \left( "getName" \right) \left( \text{TypedVals} \right) \right) \right) \right) \left( \text{TypedVals} \right) \right) \right) \right) \right) \right) \right)
\end{multlined}
\end{align*}
\]

\[
\begin{align*}
\to & \left\langle - L \mapsto - :: \text{elem} \text{StoreType} : \text{Type} \right\rangle
\end{align*}
\]

REQUIRES T \neq K \text{StoreType}

A.7.3 Array allocation

SYNTAX \( \text{DimExps} ::= \text{reverseDimExps} \left( \text{DimExps}, \text{DimExps} \right) \) [function]

RULE
\[
\text{reverseDimExps} \left( D : \text{DimExp} Ds : \text{DimExps}, Es : \text{DimExps} \right)
\]

\[
\to \text{reverseDimExps} \left( Ds, \left( D \ Es \right) \right)
\]

RULE
\[
\text{reverseDimExps} \left( \left\langle \text{DimExps}, \text{Es} : \text{DimExps} \right\rangle \right)
\]

\[
\to \text{Es}
\]

SYNTAX \( \text{DimExps} ::= \text{toDimExps} \left( \text{KListWrap}, \text{DimExps} \right) \) [function]

RULE
\[
\text{toDimExps} \left( \left[ Ds : \text{KList}, D : \text{DimExp} \right], \text{Es} : \text{DimExps} \right)
\]

\[
\to \text{toDimExps} \left( \left[ Ds \right], \left( D \ Es \right) \right)
\]

RULE
\[
\text{toDimExps} \left( \left\langle \text{KList} \right\rangle, \text{Es} : \text{DimExps} \right)
\]

\[
\to \text{Es}
\]
RULE

\[
'NewArray'(—, [ \text{D:DimExp}, \text{Ds:KList} ], \text{'List}) \quad \text{toDimExps} \left( [ \text{D, Ds} ], \text{'DimExps} \right) \quad \text{'Dims}
\]

RULE

\[
'NewArray'(—, [ \text{'List} ], —)
\]

When all dims were computed, check that dims are positive, and only after that begin array allocation.

RULE \textbf{NEWARRAY-SIMPLE}

\[
\text{new } \text{T:Type} \text{Ds:DimExps} \text{'Dims}
\]
\[
\text{checkNonNegative} (\text{Ds}) \leadsto \text{allocAndInitArray} (\text{T}, \text{reverseDimExps} (\text{Ds, 'DimExps}), \text{default} (\text{T}))
\]

\text{REQUIRES } \text{isKResult(Ds)}

[structural]

RULE \textbf{NEWARRAY-ARRAYINIT}

\[
\text{new } \text{array0f T:Type} \text{'Dims} \{ \text{InitContent:VarInits} \}
\]
\[
\text{arrayInitAlloc} (\{ \text{new } \text{T [ length(InitContent) :: int } \text{'Dims}, \{ \text{InitContent} \} \})
\]

Length of an array initializer.

SYNTAX \[ \text{Int ::= length (VarInits)} \] [function]

RULE

\[
\text{length (V:VarInit, Vs:VarInits)}
\]
\[
\text{length (Vs) +Int 1}
\]

RULE

\[
\text{length ('VarInits)}
\]
\[
\text{0}
\]

SYNTAX \[ \text{KItem ::= checkNonNegative (DimExps)} \]

RULE

\[
\text{checkNonNegative ( [ NI:Int :: — ] Remainings:DimExps)}
\]
\[
\text{checkNonNegative (Remainings)}
\]

\text{REQUIRES } \text{NI} \geq_{\text{Int}} 0
RULE checkNonNegative-to-NegativeArraySizeException
    checkNonNegative ( [ NI : Int :: — ] — )
    throw new class String2Id ("java.lang.NegativeArraySizeException") ( null :: class String ) ;
REQUIRES NI < _Int 0

RULE
    checkNonNegative ( 'DimExps )
    ’K

Same as 'NewArray, but after DimExps were computed and checked

SYNTAX  KItem ::= allocAndInitArray ( Type, DimExps, K )

Here we are actually using the reversed DimExps

RULE allocAndInitArray-MultiDim-desugar
    allocAndInitArray ( T : Type, Dim1K : DimExp Dim2K : DimExp DES : DimExps, InitExp : K )
    allocAndInitArray ( array0f T, Dim2K DES, allocAndInitArray ( T, Dim1K, InitExp ) )
    [structural]

The increment of <nextLoc> by one is required to avoid problems with empty arrays. Two empty arrays allocated one after another should have different starting locations, even if those starting locations are not used. This is required to identify them as two different objects. Their distinction could be tested by the operator ==.

RULE allocAndInitArray
    allocAndInitArray ( T : Type, [ NI : Int :: — ], InitExp : K )
    allocArray ( NI, T ) ↾ initArray ( LI + _Int 1, NI, InitExp ) ↾ arrayRef ( "" ) k
    array0f T, LI + _Int 1, NI ) :: array0f T
    LI : _Int
    LI + _Int 1
    nextLoc

Allocates the given number of elements in the store of the given Type. Used by array instantiation logic.

SYNTAX  KItem ::= allocArray ( Int, Type )

RULE allocArray
    allocArray ( N : Int, T : Type ) k
    ’Map
    LI ↦ ( ⊥ :: elem T )
    store
    LI → Field
    storeMetadata
    LI : _Int
    LI + _Int 1
    nextLoc
REQUIRES N > _Int 0
rule allocArray (0, —)

\[\text{\textasciicircum}_K\]

[structural]

Assign to each store location in the given range the value represented by the 3-rd argument. This value might be freshly computed for each element, and might alter another locations in the store, as is the case for multidim arrays. This procedure don’t change the element type in the store.

SYNTAX \( K\text{Item} ::= \text{initArray} \ (\text{Int}, \text{Int}, \text{K}) \)

RULE initArray

\[
\text{initArray} \ (\text{OL}:\text{Int}, \text{N}:\text{Int}, \text{InitExp}:\text{K})
\]

\[
\text{store} \ (\text{OL}, \text{InitExp}) \bowtie \text{initArray} \ (\text{OL} +_{\text{Int}} 1, \text{N} -_{\text{Int}} 1, \text{InitExp})
\]

REQUIRES \( \text{N} >_{\text{Int}} 0 \)

[structural]

RULE

\[
\text{initArray} \ (\text{—}, 0, 
\text{—})
\]

\[\text{\textasciicircum}_K\]

[structural]

### A.7.4 Array initializer

Allocates the array based on previously computed size, then proceeds to array initialization

SYNTAX \( K\text{Item} ::= \text{arrayInitAlloc} \ (\text{K}, \text{ArrayInit}) \) [strict(1)]

RULE arrayInitAlloc

\[
\text{arrayInitAlloc} \ (\text{arrayRef} \ (\text{T}:\text{Type}, \text{L}:\text{Int}, \text{Len}:\text{Int}) :: \text{T}, \{ \text{InitContent}:\text{VarInits} \})
\]

\[
\text{arrayInitImpl} \ (\text{T}, \text{L}, \{ \text{InitContent} \}) \bowtie \text{arrayRef} \ (\text{T}, \text{L}, \text{Len}) :: \text{T}
\]

SYNTAX \( K\text{Item} ::= \text{arrayInitImpl} \ (\text{Type}, \text{Int}, \text{ArrayInit}) \)

RULE arrayInitImpl-not-ArrayInit

\[
\text{store} \ (\text{L}, \text{E}) \bowtie \text{arrayInitImpl} \ (\text{—}, \text{L}:\text{Int}, \{ \text{E}:\text{Exp}, \text{Remaining}:\text{VarInits} \})
\]

\[
\text{L} +_{\text{Int}} 1 \quad \{ \text{Remaining} \}
\]
A.8 Module NEW-INSTANCE

A.8.1 Background

In this subsection we present the fragment of configuration used by runtime method invocation. The figure below contains the cells and their sorts.
The cell \( \{ k \} \) stores the current computation. Inside \( \{ \text{env} \} \) we store the local environment — a map from variable names to their locations in the store. The cell \( \{ \text{methodContext} \} \) store information about the current object — the one accessible through the keyword this. Both \( \{ \text{env} \} \) and \( \{ \text{methodContext} \} \) play a special role in object instantiation.

The cell \( \{ \text{class} \} \) contains various sub-cells holding the content of that class. The first cell in \( \{ \text{classType} \} \) 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 \( \{ \text{class} \} \). Next relevant cells inside \( \{ \text{class} \} \) are \( \{ \text{enclosingClass} \} \) — the directly enclosing class in case this class is an inner class. The vase class is stored inside \( \{ \text{extends} \} \) and the list of declarations of instance fields without identifiers is stored in \( \{ \text{instanceFields} \} \).

The next two cells are related to the store. The cell \( \{ \text{store} \} \) has a central role in the semantics — it is the map from object locations (values in the cell \( \{ \text{env} \} \)) to their actual typed values. The cell \( \{ \text{nextLoc} \} \) is the counter of store locations.

The remaining big group of cells — \( \{ \text{objectStore} \} \) contains the inner structure of objects. The \( \{ \text{objectId} \} \) is an unique identifier of the object. Every reference to this object in the store is a reference to this id. Inside \( \{ \text{objectType} \} \) is the actual runtime type of the object. Next we have a list of \( \{ \text{layer} \} \) 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 \( \{ \text{layerClass} \} \) stores its associated class, \( \{ \text{layerEnv} \} \) — the fields and \( \{ \text{layerEnclosingObject} \} \) — the enclosing object, in the case when \( \{ \text{layerClass} \} \) 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.

### A.8.2 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

\[
\text{Qual} \cdot \text{new} \quad \text{Name} \vdash id \\
\text{createClassType ( toPackage ( typeOf ( Qual) ), Name)} \\
\]

After the new instance expression has 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 \( \text{nextLoc} \) is used as the location of the newly created object. The counter is incremented for the next use. Inside \( \text{objectStore} \) a new cell \( \text{object} \) is created for the new object. For now it has just two sub-cells specified — \( \text{objectId} \) and \( \text{objectType} \), and no layers. Curiously we don’t have to specify neither \( \text{object} \) nor \( \text{objectStore} \) 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\[43\]. In the cell \( \text{store} \) a new entry is created with key being \( L \) and value — a reference to the newly created object in \( \text{object} \). The content of \( \text{methodContext} \) is reset to a default state. This default state is required by rules that are applied next.

Inside \( \text{k} \) the new instance expression is rewritten into a sequence of computations that will be executed by the following rules. The auxiliary function \text{staticInit()}\) triggers static initialization of the instantiated class, in case it was not triggered earlier. Next, the function \text{create()}\) populates the layers of the object inside \( \text{object} \) This also includes allocation of all instance fields, and their initialization to the default value. Field initializers are not executed yet. The function \text{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 \( \text{layer} \) will be set as part of constructor invocation.

The next term in the computation (the one starting with \text{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 \text{typedLookup(L)} \) is evaluated into the object stored at the location \( L \), that will serve as a qualifier for constructor invocation. The function \text{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 \text{restoreMethoContext()} \) restores \( \text{methodContext} \) to the the state before object creation. The last term is the result value of the object instantiation expression.
RULE QUALIFIED-NEW-INSTANCE

\[
\begin{align*}
\text{(Qual:KResult . new Class:ClassType (Args:TypedVals))}
& \quad \text{staticInit (Class) } \leadsto \text{create (Class) } \leadsto \text{setEncloser (typedLookup (L), Class, Qual) } \leadsto \text{(typedLookup (L). getConsName (Class)(Args); ) }
& \quad \leadsto \text{restoreMethContext (MethContext) } \leadsto \text{typedLookup (L)}
\end{align*}
\]

\[
\begin{align*}
\text{store } & \quad \frac{L: Int}{L + Int 1} \text{nextLoc}
\text{storeMetadata}
\end{align*}
\]

SYNTAX \[KItem ::= \text{create (ClassType)}\]

RULE CREATE

\[
\begin{align*}
\text{create (Class)}
& \quad \text{create (BaseClass) } \leadsto \text{setCrntClass (Class) } \leadsto \text{FieldDecs } \leadsto \text{addEnvLayer}
\end{align*}
\]

\[
\begin{align*}
\text{methodContext}
\end{align*}
\]

RULE CREATE-EMPTY-Discard

\[
\begin{align*}
\text{create (\textit{\`K})}
\end{align*}
\]

\[
\begin{align*}
\text{[structural]}
\end{align*}
\]

SYNTAX \[KItem ::= \text{setCrntClass (ClassType)}\]

RULE SETCRNTCLASS

\[
\begin{align*}
\text{setCrntClass (Class)}
& \quad \text{'\`K}
\end{align*}
\]

\[
\begin{align*}
\text{[structural]}
\end{align*}
\]

SYNTAX \[KItem ::= \text{addEnvLayer}\]
RULE addEnvLayer

\[ \text{addEnvLayer} ( K, \text{Env} \mapsto \text{Class}) \mapsto \text{crntClass} \mapsto \text{OID} \mapsto \text{location} \]

\[ \text{OID} \mapsto \text{objectld} \mapsto \text{Bag} \mapsto \text{object} \]

[structural]

Sets the enclosing object for a given object.

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

RULE setEncloser-value

\[ \text{setEncloser} ( \text{objectRef} (\text{OID}, --) :: --, \text{Class}, \text{EncloserVal} :: --) \mapsto K \]

\[ \text{Class} \mapsto \text{layerClass} \mapsto -- \mapsto \text{EncloserVal} :: \text{EncloserClass} \mapsto \text{layerEnclosingObject} \mapsto \text{Class} \mapsto \text{classType} \]

\[ \text{EncloserClass} \mapsto \text{enclosingClass} \]

RULE setEncloser-noValue

\[ \text{setEncloser} (--, --, K) \]

\[ K \]

A.9 Module NEW-INSTANCE-REST

Additional semantics of new instance creation.

A.9.1 Instance field declarations

RULE FieldDec-instance

\[ \text{FieldDec}( [\text{KList}], \text{T}\mapsto \text{Type}, [\text{VarDec(X:Id)}]) \mapsto K \]

\[ \text{Env}\mapsto \text{Map} \mapsto \text{Environement} [\text{L} / \text{X}] \]

\[ \text{L} \mapsto \text{default}\ (T) \mapsto \text{store} \mapsto \text{Map} \mapsto \text{storeMetadata} \]

\[ \text{L} + \text{Int} \mapsto \text{nextLoc} \]

159
A.9.2 Explicit constructor invocation — this() and super()

**RULE QSuperConstrInv**

\[
\text{QSuperConstrInv(} \text{Qual:}K, \quad \text{—, [ } \text{Args:}K\text{List }\text{]}) \quad \xrightarrow{\text{setEncloser}} \quad \text{}'QThis(Class), \text{ BaseClass, Qual} \quad \xrightarrow{\text{'Invoke'(QSuperMethod(Class, }} \quad \cdot K, \text{ getConsName(BaseClass)}, \text{ [ } \text{Args }\text{]) ; ))} \quad \xrightarrow{\text{Class:}ClassType} \quad \text{crntClass} \quad \langle \text{Class} \rangle \text{classType} \quad \langle \text{BaseClass:}ClassType \rangle \text{extends}
\]

**RULE AltConstrInv**

\[
\text{AltConstrInv(} \quad \text{—, [ } \text{Args:}K\text{List }\text{])} \quad \xrightarrow{\text{'}QThis(Class), \text{ getConsName(Class)( toExps ( [ } \text{Args }\text{])) ; }} \quad \xrightarrow{\text{Class:}ClassType} \quad \text{crntClass}
\]

A.10 Module METHOD-INVOKE

A.10.1 Background

In this section we present the fragment of configuration used by runtime method invocation. The figure below contains the cells and their sort\[^1\]

The cell \(\{\}_k\) stores the current computation. The cell \(\{\}_\text{stack}\) is a list of pairs of the form \((K, \text{Bag})\), 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 \text{Bag} represents the content of cell \(\{\}_\text{methodContext}\) at the moment of method call.

The cell \(\{\}_\text{class}\) contains various sub-cells holding the content of that class. The first cell in \(\{\}_\text{classType}\) of sort \text{ClassType} that holds the fully qualified class name. This cell serves as a key in all rules that match a

\[^1\]AM*** = AccessMode, CT*** = ContextType
fragment of a \[
\langle\text{class}\rangle
\]
. The value in the cell \[
\langle\text{classMetaType}\rangle
\]
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 \[
\langle\text{methods}\rangle
\]. 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 \[
\langle\text{methods}\rangle
\] 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 \[
\langle\text{methodDec}\rangle
\] represents a method declared inside the current class. The subcell \[
\langle\text{methodSignature}\rangle
\] 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:

**SYNTAX**

\[
\text{KResult ::= ClassType} \\
| \text{TypedVal}
\]

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:

**SYNTAX**

\[
\text{TypedVal ::= objectRef (Int, ClassType) :: ClassType} \\
| \text{null :: ClassType}
\]

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 will also consider \[\cdot\] — the unit element of K to be KResult. The value \[\cdot\] is often used in auxiliary functions as a placeholder until some actual value is computed.

### A.10.2 Introduction

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

- An invocation of a static method qualified by its class: \text{Class.f(args)}

- An invocation of a static method qualified by an expression producing an object: \text{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 \text{o; class.f(args)}; either, because \text{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: \text{o.f(args)}

- An invocation of an instance method qualified by an interface reference: \text{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

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.

### A.10.3 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** 

```
Exp ::= K . MethodName (Exps) [seqstrict(1,3)]
```

**SYNTAX** 

```
MethodName ::= Id
```

**SYNTAX** 

```
Exps ::= List{Exp ,"","} [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.

A.10.4 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 `⟨⟨⟨⟩⟩⟩` class and `⟨⟨⟨⟩⟩⟩` classDec. The rules from step 3 simply load the relevant information from configuration cells to `methodInfo()` arguments.

**SYNTAX**

\[
\text{MethodName ::= methodInfo ( Signature, RefType, ClassMetaType, RefType, ContextType, AccessMode )}
\]

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**

\[
\text{Invoke-compute-methodInfo-Signature}
\]

\[
\text{Qual:KResult}.
\]

\[
\text{methodInfo ( sig (Name, getTypes (Args)), typeOf (Qual), \text{Acc})
\]

\[
\text{Args:TypedVals)}
\]

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:

**SYNTAX**

\[
\text{TypedVals ::= List\{TypedVal , “,”\}}
\]

163
SYNTAX  \( \text{Exps ::= TypedVals} \)

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

The second rule for method invocation loads \( \text{MetaT} \) and \( \text{DecC} \). It requires \( \text{Sig} \) and \( \text{QualC} \) computed by the previous rule.

**RULE \text{Invoke-compute-methodInfo-DecC}**

\[
\begin{align*}
\text{methodInfo} \left( \text{Sig}\cdot\text{Signature}, \text{QualC}\cdot\text{ClassType}, \text{MetaT}, \text{DecC} \right)
\end{align*}
\]

\[
\begin{align*}
\{ \text{QualC} \}_{\text{classType}} \{ \text{MetaT}\cdot\text{ClassMetaType} \}_{\text{classMetaType}} \{ \text{Methods}\cdot\text{Map} \}_{\text{methods}}
\end{align*}
\]

There is one case that is not covered by the previous rule — the case when the cell \( \{ \text{methods} \} \) does not have a key equal to \( \text{Sig} \). 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 \( \text{QualT} \). For this case \( \text{DecC} \) cannot be computed, but we know for sure that \( \text{ContextT} \) for an abstract method is instance. Also, because the method was declared in an interface, it is certainly public.

**RULE \text{Invoke-compute-methodInfo-unmapped-method-ContextType}**

\[
\begin{align*}
\text{methodInfo} \left( \text{Sig}\cdot\text{Signature}, \text{QualC}\cdot\text{ClassType}, \text{MetaT}, \text{instance}, \text{public} \right)
\end{align*}
\]

\[
\begin{align*}
\{ \text{QualC} \}_{\text{classType}} \{ \text{MetaT}\cdot\text{ClassMetaType} \}_{\text{classMetaType}} \{ \text{Methods}\cdot\text{Map} \}_{\text{methods}}
\end{align*}
\]

**REQUIRES** \( \neg \text{Bool} \) \( \text{Sig} \) in keys \( \{ \text{Methods} \} \)

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

**RULE \text{Invoke-compute-methodInfo-ContextType}**

\[
\begin{align*}
\text{methodInfo} \left( \text{Sig}\cdot\text{Signature}, \text{DecC}\cdot\text{ClassType}, \text{ContextT}, \text{Acc} \right)
\end{align*}
\]

\[
\begin{align*}
\{ \text{DecC} \}_{\text{classType}} \{ \text{Sig} \}_{\text{methodSignature}} \{ \text{ContextT}\cdot\text{ContextType} \}_{\text{methodContextType}}
\end{align*}
\]

\[
\begin{align*}
\{ \text{Acc}\cdot\text{AccessMode} \}_{\text{methodAccessMode}}
\end{align*}
\]

**A.10.5 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 \text{methodInfo}() were filled in (where possible). The rules of this step rewrite \text{methodInfo}() into \text{methodRef}() — another auxiliary data structure. The production \text{methodRef}() is a
reference to a method declaration. It contains two fields — \texttt{Sig} and \texttt{DecC} — the signature and the declaration class. The implementation class is the class that contains the actual method declaration to be invoked.

**SYNTAX**

\[
\text{MethodName} ::= \text{methodRef} (\text{Signature, RefType})
\]

Since we already know the signature, this phase amounts to computing \texttt{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 \texttt{DecC} is the qualifying type. The qualifier is discarded by rewriting it into \texttt{\_\_K}.

**RULE Invoke-methodInfo-static**

\[
\begin{array}{l}
\text{methodInfo (Sig:Signature, \_\_K, class, DecC:ClassType, static, \_)}(\_)
\end{array}
\]

If the qualifier value is null and \texttt{ContextT} is instance, then \texttt{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 \texttt{ContextT} is static, then the previous rule will match, and no exception will be thrown.

**RULE Invoke-methodInfo-instance-on-null**

\[
\begin{array}{l}
\text{null :: \_\_K. methodInfo (\_\_K, \_\_K, \_\_K, \_\_K, instance, \_)}(\_)
\end{array}
\]

\[
\text{throw new NullPointerException (null :: String)};
\]

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**

\[
\begin{array}{l}
\text{objectRef (\_\_K, \_\_K) :: \_\_K. methodInfo (Sig:Signature, \_\_K, class, DecC:ClassType, instance, private)}(\_)
\end{array}
\]

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 (\texttt{ObjC}). Recall that the runtime type of an object is stored in the second argument of \texttt{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 \texttt{Sig} in the cell \texttt{\_\_K.methods} associated with the actual object class. This is because the unfolding phase populates \texttt{\_\_K.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 \texttt{DecC} is the class where the right method version is declared.
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 `lookupPackageMethod()`. 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 `%K`.

**Rule `Invoke-methodInfo-instance-package`**

\[
\text{objectRef} \quad \text{::=} \quad (\text{--}, \text{ObjC}:\text{ClassType}) \quad :: \quad \text{QualC}:\text{ClassType}
\]

\[
\text{methodInfo} \quad (\text{Sig}:\text{Signature}, \text{QualC}, \text{class}, \text{--}, \text{instance}, \text{package}) \quad \text{::=} \quad \text{دقية} \quad (\text{--}) \quad \cdot
\]

\[
\text{lookupPackageMethod} \quad (\text{Sig}, \text{getClassChain} \quad (\text{QualC}, \text{ObjC}), \%K)
\]

\[
\text{QualC} \text{classType} \quad (\%\text{Sig} \rightarrow \text{--} \cdot \text{methods}
\]

Before the evaluation of `lookupPackageMethod()` may begin, the term `lookupPackageMethod()` 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 `lookupPackageMethod()`. If it has other forms, such as the method name or `methodInfo()`, it won’t be heated.

**CONTEXT**

\[
\text{--}:\text{K} \quad \square (\text{--})
\]

**REQUIRES** `getKLabel (\square) = \text{KLabel 'lookupPackageMethod}

Returns the list of classes representing the layer of the given object (by OID), located between MostBase-Class (exclusively) and MostDerivedClass (inclusively).

**SYNTAX**  

\[
\text{KItem ::= =addClassChain ( ClassType, ClassTypes )}
\]

The right signature is already found. Search for the right implementation is performed from the compile-time type of the target to more derived types, up to the object type of the target. This is required in order to respect the rules of inheritance in the presence of access modes. Evaluates into typed method closure.

**SYNTAX**  

\[
\text{KItem ::= =lookupPackageMethod ( Signature, ClassTypes, K ) [strict(2,3)]}
\]

The rules for `lookupPackageMethod()` are based on the following two properties of the configuration:
• if the cell \( \text{methods} \) for a particular class contains a key \( \text{Sig} \), then \( \text{methods} \) for all classes derived from it will contain the key \( \text{Sig} \).

• if a particular class contain a method declaration with signature \( \text{Sig} \) access mode \( \text{Acc} \), then all declarations of \( \text{Sig} \) in derived classes (that are not necessarily overriding!) will have the access mode equal to either \( \text{Acc} \) or a value wider than \( \text{Acc} \).

The first property is ensured by the unfolding algorithm. Because \( \text{methods} \) of a derived class inherit all the \( \text{methods} \) of the direct base class, the map \( \text{methods} \) 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 (\( \text{CurrentC} \)), and possibly rewrites the third argument into the current class. The first rule matches when there is no declaring class yet (third argument is \( \cdot \), the initial case).

\[
\text{rule lookupPackageMethod-layer-first-dec-found}
\begin{align*}
\text{lookupPackageMethod} \left( \text{Sig:Signature}, \text{CurrentC:ClassType}, \text{Cs:ClassTypes}, \cdot \to \text{DecC:ClassType} \right)
\end{align*}
\]

The second rule matches when we already found a declaring class \( \text{OldDecC} \) and the current class \( \text{CurrentC} \) has another method declaration with the right signature. The presence of a declaration with signature \( \text{Sig} \) inside \( \text{CurrentC} \) is identified by the match \( \text{CurrentC:ClassType} \to \text{Sig} \to \text{CurrentC:ClassType} \to \text{methods} \), according to the definition of \( \text{methods} \).

If the method in \( \text{CurrentC} \) directly overrides the method in \( \text{OldDecC} \), the declaring class is updated to \( \text{CurrentC} \). Otherwise the declaring class stays unchanged. The rules for direct overriding (case 1a above) are defined in the auxiliary function \( \text{isOverridden()} \). The function takes 3 arguments:

• The base class \( \text{OldDecC} \)
• The derived class \( \text{CurrentC} \)
• The access mode \( \text{Acc} \) of the definition of \( \text{Sig} \) in \( \text{OldDecC} \).

\[
\text{rule lookupPackageMethod-new-method}
\begin{align*}
\text{lookupPackageMethod} \left( \text{Sig:Signature}, \text{CurrentC:ClassType}, \text{Cs:ClassTypes}, \cdot \right)
\end{align*}
\]

Tests if a method declared in class \( \text{BaseC} \) with access mode \( \text{Acc} \) is overridden by a method with the same signature declared in a subclass \( \text{SubC} \).

The method is overridden if either:
• Acc is protected or public
• Acc is package and BaseC and SubC are declared in the same package (JLS §6.6)

**SYNTAX** \( KItem ::= \text{isOverridden}(\text{ClassType, AccessMode, ClassType}) \)

**RULE**

\[
\text{isOverridden}(\text{—, public, —)} \quad \Rightarrow \quad \text{true}
\]

**RULE**

\[
\text{isOverridden}(\text{—, protected, —)} \quad \Rightarrow \quad \text{true}
\]

**RULE**

\[
\text{isOverridden}(\text{BaseC:ClassType, package, SubC:ClassType}) \quad \Rightarrow \\
\text{eqAux (getPackage(getTopLevel(BaseC)), getPackage(getTopLevel(SubC)))}
\]

**RULE**

\[
\text{isOverridden}(\text{—, private, —)} \quad \Rightarrow \quad \text{false}
\]

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 \( \text{CurrentC} \neq K \text{DecC} \). Indeed, the two classes are different only when the entry \( \text{Sig} \mapsto \text{DecC} \) in \( \text{methods} \) was inherited rather than produced by a method in CurrentC.

**RULE** lookupPackageMethod-no-new-method

\[
\begin{align*}
\langle \text{lookupPackageMethod}(\text{Sig:Signature, CurrentC:ClassType, Cs:ClassTypes, —, —}) \rangle_{CS} \\
\langle \text{CurrentC} \rangle_{\text{classType}} \langle \text{ — Sig} \mapsto \text{DecC:ClassType} \rangle_{\text{methods}}
\end{align*}
\]

**REQUIRES** \( \text{CurrentC} \neq K \text{DecC} \)

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

**RULE** lookupPackageMethod-end

\[
\text{lookupPackageMethod}(\text{Sig:Signature, 'C:ClassTypes, DecC:ClassType}) \\
\text{methodRef (Sig, DecC)}
\]

For the code example above, the term \( \text{lookupPackageMethod()} \) will pass through the following forms during evaluation:
A.10.6 Actual method invocation

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

- saves the rest of computation (RestK) and the content of \langle\langle\langle\text{methodContext}\rangle\rangle\rangle as a new entry of the cell \langle\langle\langle\text{stack}\rangle\rangle\rangle. This data is restored back by the rules for return statement.
- Initializes the new method context.
  - The local variable environment \langle\langle\langle\text{env}\rangle\rangle\rangle is emptied
  - current class \langle\langle\langle\text{crntClass}\rangle\rangle\rangle is initialized to the class declaring the method
  - object location \langle\langle\langle\text{location}\rangle\rangle\rangle is initialized to the location of the qualifier object for instance methods, or \texttt{\_K} for static methods. The extraction of the location from the qualifier value is performed by the function \texttt{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 \texttt{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**

\[
\begin{align*}
\text{Qual} & \to KResult \\
\text{methodRef} & \to (\text{Sig} : \text{Signature}, \ DecC : \text{ClassType}) (\ Arg : \text{TypedVals}) \\
\text{staticInit} & \to (\ DecC ) \\
& \to \text{initParams} (\ Params, \ Args ) \\
& \to \text{Body} \to \text{return} (\ '\text{Some(}\
\text{nothing :: void }) ) ; \\
& \to \text{'List} \\
& \to \text{stack} \\
& \to \text{MethodContext} : \text{Bag} \\
& \to \text{env} \to \text{DecC} \to \text{crntClass} \to \text{getOId} (\ \text{Qual}) \to \text{location} \\
& \to \text{methodSignature} \to \text{Params} : \text{Params} \\
& \to \text{methodParams} \to \text{Body} : \text{K} \to \text{methodBody} \\
\text{REQUIRES} & \to \text{getKLabel} (\ \text{Body} ) \neq \text{KLabel} '\text{NoMethodBody}
\end{align*}
\]

**SYNTAX**  

\[
\text{KItem} ::= \text{getOId} (\ K ) [\text{function}]
\]

**RULE**

\[
\begin{align*}
\text{getOId} & \to \text{objectRef} (\ \text{OId} : \text{Int}, \ --- ) \to --- \\
& \to \text{OId}
\end{align*}
\]

**RULE**

\[
\begin{align*}
\text{getOId} & \to \ 'K \\
& \to \ 'K \\
\end{align*}
\]

Binds a list of formal parameters to their values. Used by method calls and try-catch.

**SYNTAX**  

\[
\text{KItem} ::= \text{initParams} (\ \text{Params} , \ \text{TypedVals} )
\]

**RULE INITPARAMS**

\[
\begin{align*}
\text{initParams} & \to (\ T : \text{Type}, \ X : \text{Id}, \ RestP : \text{Params} , (\ TV : \text{TypedVal}, \ RestV : \text{TypedVals} )) \\
& \to (\ TX ; ) \to (\ X = ( (\ T ) : \text{TV} : \text{TypedVal} )) ; \to \text{initParams} (\ RestP , \ RestV )
\end{align*}
\]

171
RULE initParams-end

initParams ('Params', 'TypedVal')

'K

[structural]

A.10.7 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.
A.11 Module METHOD-INVOKE-REST

The module contains additional semantics related to method invocation.

A.11.1 Method information for arrays and strings

The next rule for method invocation is applied when the qualifier type is array. This array type is rewritten into the auxiliary class `ArrayImpl`, that is used in K-Java to simulate method invocations over array objects. This rule is required in order to minimize the number of particular cases involving arrays in the rules that follow.

```plaintext
RULE Invoke-compute-methodInfo-arrays

QualRV::RawVal : arrayOf T:Type . classArrayImpl . methodInfo (Sig:Signature, arrayOf T, classArrayImpl, —, —, —, —)

```

```plaintext
RULE Invoke-methodInfo-on-array-or-string

QualRV::RawVal :: QualC:ClassType . methodInfo (Sig:Signature, QualC, class, DecC:ClassType, instance, —) (Args:TypedVals)

invokeImpl (methodRef (Sig, DecC), QualRV :: QualC, toKListWrap (Args))

REQUIRES isArrayRef(QualRV) = K true \lor \text{Bool isString(QualRV)} = K true
```

A.11.2 Superclass method access — A.super(...) 

```plaintext
RULE Invoke-QSuperMethod

'Invoke( QSuperMethod(Class:ClassType, —:K, Name:Id) , —)

'Method('MethodName(superMethodImpl ('QThis(Class)), Name))

```

Elaboration result for super keyword in A.super.m() call. Have to be a typed expression.

**SYNTAX**  
`MethodName ::= superMethodImpl (K) [strict]`

```plaintext
RULE superMethodImpl

superMethodImpl (objectRef (OID:Int, —) :: Class:ClassType) `k` (Class)classType

objectRef (OID, BaseClass) :: BaseClass

<BaseClass:ClassType> extends
```
A.11.3 Auxiliary functions

**RULE getClassChain-process**

\[
\text{getClassChain ( MostBaseClass:ClassType, MostDerivedClass:ClassType, Cs:ClassTypes )} \rightarrow_k \text{MostDerivedClass } \text{classType}
\]

\[
\text{BaseClass, MostDerivedClass, Cs} \text{extends}
\]

**REQUIRES** MostDerivedClass \( \neq_K \) MostBaseClass

**RULE getClassChain-end**

\[
\text{getClassChain ( Class:ClassType, ( Class, MoreDerivedClasses:ClassTypes ) )}
\]

\[
( \text{Class, MoreDerivedClasses})
\]

**RULE Invoke-methodRef-native**

\[
\text{Qual:KResult . methodRef ( sig ( Name:Id, Ts:Types), DecC:ClassType) ( Args:TypedVals)} \rightarrow_k \text{DecC } \text{classType} \text{sig ( Name, Ts)methodSignature } \text{’NoMethodBody(—) } \text{methodBody}
\]

**REQUIRES** \( \neg \text{Bool (( DecC } =_K \text{class Object) } \land \text{Bool ( Name } =_K \text{String2Id ("wait") } \lor \text{Bool Name } =_K \text{String2Id ("notify") } \lor \text{Bool Name } =_K \text{String2Id ("notifyAll") }) \)

Auxiliary function for methods that need implementation in the semantics. The implementation of this production is given in api-core.k and api-threads.k.

**SYNTAX**

\[
KItem ::= \text{invokeImpl ( MethodRef, K, KListWrap)} \ [\text{strict(1)}]
\]

A.12 Module STATIC-INIT

Triggers the static initialization of a class. The detailed initialization procedure is described in JLS §12.4.2. Here we implement an approximation of that algorithm, one that does not use the traditional Java synchronization mechanism, but relies instead on term rewriting capabilities to implement locks. This algorithm is required to avoid multithreaded issues during static initialization. This may happen when two threads access an uninitialized class at the same time.

Initialization status of a class may have one of the following 3 states: StaticUninitialized, StaticInitializing(TId), StaticInitialized. Here TId is the identifier of the thread that initiated the initialization process. When we reach staticInit(Class), depending on the initialization status of the class (stored in \( \langle \rangle \text{staticInitStatus} \)), we should do one of the following:

- **status = StaticUninitialized** — Perform the initialization.
- **status = StaticInitializing(TId), TId = this thread** — This is a recursive initialization request, discard the term.
• status = StaticInitializing(TId), TId = another thread — We should wait until the initialization is completed by another thread. Do not match this case.
• status = StaticInitialized — The class has already been initialized. Discard the term.

**SYNTAX**

\[ KItem ::= \text{staticInit} \ ( \text{ClassType} ) \]

**RULE STATICINIT**

\[
\text{staticInit} \ ( \text{Class:ClassType} )
\]

\[
\left[ \text{StaticFields}, \text{staticInit} \ ( \text{BaseClass} ) \right], \ 'Try'(\text{StaticInit}, \ ['Catch'(\text{ParamImpl}(
\text{class Object, String2Id } "e")), \ 'Block'(\text{throw new class String2Id (}
"java.lang.ExceptionInInitializerError")((\text{class Object}) \text{String2Id } "e")));}), \]

\[
\left[ \text{restoreMethContext} ( \text{MethContext} ), \text{staticInitDone} ( \text{Class} ) \right] \]

**RULE STATICINIT-ALREADY-INITIALIZED-DISCARD**

\[
\text{staticInit} \ ( \text{Class:ClassType} )
\]

\[
\left[ TId: \text{Int} \right] \text{tid} \ ( \text{Class} ) \text{classType}
\]

**RULE STATICINIT-RECURSIVE-INITIALIZING-DISCARD**

\[
\text{staticInit} \ ( \text{Class:ClassType} )
\]

\[
\left[ TId: \text{Int} \right] \text{tid} \ ( \text{Class} ) \text{classType}
\]

**RULE STATICINIT-EMPTY**

\[
\text{staticInit} \ ( \ 'K' \ )
\]

Triggers the last step of static initialization: static init status transitions from StaticInitializing to StaticInitialized.

**SYNTAX**

\[ KItem ::= \text{staticInitDone} \ ( \text{ClassType} ) \]
RULE staticInitDone

\[
\begin{align*}
\text{rule staticInitDone} & \quad (\text{Class}:\text{ClassType}) \\
\text{staticInitDone} & \quad (\text{TId}:\text{Int}) \quad \text{tid} \quad (\text{Class}:\text{classType}) \\
\text{StaticInitializing} & \quad (\text{TId}) \\
\text{StaticInitialized} & \quad \text{staticInitStatus}
\end{align*}
\]

RULE FieldDec-static

\[
\begin{align*}
\text{rule FieldDec-static} & \quad \langle '\text{FieldDec}([ '\text{Static}(-) ], \text{T}:\text{Type}, [ '\text{VarDec}(\text{X}:\text{Id}) ] \rangle \\
\text{FieldDec} & \quad (\text{TId}:\text{Int}) \quad \text{tid} \quad (\text{Class}:\text{classType}) \\
\text{crntClass} & \quad \text{classType} \\
\text{Env}:\text{Map} & \quad \text{staticEnv} \\
\text{L} & \mapsto \text{default}(\text{T}) \quad \text{store} \\
\text{L} & \mapsto \text{Field} \quad \text{storeMetadata} \quad \langle \text{L}:\text{Int}, \text{L}^+\text{Int} \rangle \quad \text{nextLoc}
\end{align*}
\]

A.13 Module API-CORE

This module contains the minimal part of Java API required to perform console read/write operations, as well as a few other API functions mentioned in JLS.

All the methods defined in this module through invokeImpl are native, their body is \text{'NoMethodBody(\_)}\text{', thus they cannot be matched by rules in METHOD-INVOKE.

A.13.1 System.in, System.out, Scanner

system-out-print-convert heat argument

\text{context}

\text{invokeImpl ( methodRef ( sig (MethodName:Id, \_), Class:ClassType), \_, \_ )}

\text{requires Class =}\text{K class String2Id ("java.io.PrintWriter") \& BoolId2String (MethodName) \text{==String "print"}}

System.out.print(). For integers and strings, print their value. For classes, print class type.
RULE SYSTEM-OUT-PRINT-STRING

\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), —, [ Str:String :: — ] )}
\]

nothing :: void

REQUIRES Class \(=_{K}\) class String2Id ("java.io.PrintWriter") \(\land\) Bool Id2String (MethodName) ==String "print"

[transition-threading]

Is only used for primitive type arguments. Object arguments are converted by Java code inside the class PrintWriter.

RULE SYSTEM-OUT-PRINT-CONVERT

\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), —, [ RV:RawVal :: T:Type ] )}
\]

toString ( RV :: T )

REQUIRES Class \(=_{K}\) class String2Id ("java.io.PrintWriter") \(\land\) Bool ( Id2String (MethodName) ==String "print" ) \(\land\) Bool \(\neg\) Bool ( isString(RV) \(=_{K}\) true )

Scanner.nextInt(), used to read from the console.

RULE SCANNER-NEXTINT

\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), —, [ \_KList ] )}
\]

readInt

REQUIRES Class \(=_{K}\) class String2Id ("java.util.Scanner") \(\land\) Bool Id2String (MethodName) ==String "nextInt"

RULE SCANNER-NEXTLINE

\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), —, [ \_KList ] )}
\]

readString

REQUIRES Class \(=_{K}\) class String2Id ("java.util.Scanner") \(\land\) Bool Id2String (MethodName) ==String "nextLine"

An intermediate construct for reading from console, meant mostly to simplify debugging.

SYNTAX \(\text{KItem ::= }\text{readInt}

| \text{readString}\)

RULE READINT

\[
\begin{array}{c}
\text{readInt} \\
\text{I :: int}
\end{array}
\]

\[
\begin{array}{c}
\text{in}
\end{array}
\]

177
A.13.2 Class Object

Object.getClass()

invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), objectRef ( OL:Int, —) :: —, [ 'KList ] )

'Lit('Class(BottomClass))

store 〈 OId-objectId 〉

REQUIRES Class = K \text{class Object} \land \text{Bool} \text{Id2String} \text{MethodName} == \text{String} "\text{getClass}"

invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), Str:String :: —, [ 'KList ] )

'Lit('Class(class String))

REQUIRES Class = K \text{class Object} \land \text{Bool} \text{Id2String} \text{MethodName} == \text{String} "\text{getClass}"

invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), arrayRef ( ArrT:Type, —, —) :: —, [ 'KList ] )

'Lit('Class(ArrT))

REQUIRES Class = K \text{class Object} \land \text{Bool} \text{Id2String} \text{MethodName} == \text{String} "\text{getClass}"

Object.toString(). For Object.toString() the implementation is in the library Java code.

invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), Str:String :: —, [ 'KList ] )

Str :: class String

REQUIRES Class = K \text{class Object} \land \text{Bool} \text{Id2String} \text{MethodName} == \text{String} "\text{toString}"

Object.hashCode(). Returns the canonical memory location of the objectRef.
RULE object-hashCode
\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), objectRef (}
\text{OL:Int, —) :: —, [KList ])}
\]
\[
\text{OId :: int}
\]
\[
\langle - OL \mapsto objectRef (OId:Int, —) :: — \rangle_{\text{store}}
\]
REQUIRES Class =_{K} class Object \land \text{Bool Id2String (MethodName) }=\text{String "hashCode"}

A.13.3 Class String

int String.length()

RULE String-length
\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), Str:String :: Class,}
\]
\[
\text{[KList ]}
\]
\[
\text{lengthString (Str)}
\]
REQUIRES Class =_{K} class String \land \text{Bool Id2String (MethodName) }=\text{String "length"}

RULE String-charAt
\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), Str:String :: Class,}
\]
\[
\text{[ I:Int :: int ]}
\]
\[
\text{ordChar ( substrString (Str, I, I +Int 1)) :: char}
\]
REQUIRES Class =_{K} class String \land \text{Bool Id2String (MethodName) }=\text{String "charAt"}

static native String String.valueOf(int i);

RULE String-valueOf
\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), Str:String :: Class,}
\]
\[
\text{[ I:Int :: int ]}
\]
\[
\text{Int2String (I)}
\]
REQUIRES Class =_{K} class String \land \text{Bool Id2String (MethodName) }=\text{String "valueOf"}

boolean String.equals(Object)

RULE String-equals
\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), Str:String :: Class,}
\]
\[
\text{[ ParamStr:String :: — ]}
\]
\[
'{Eq(Str, ParamStr)}
\]
REQUIRES Class =_{K} class String \land \text{Bool Id2String (MethodName) }=\text{String "equals"}

String.toString()
RULE STRING-toString

invokeImpl (methodRef (sig (MethodName:Id, —), Class:ClassType), Str: String :: Class, [‘KList ]

Str

REQUIRES Class =_{K} class String ∩_{Bool} Id2String(MethodName) == String "toString"

int String.compareTo(Object another)

RULE STRING-compareTo

invokeImpl (methodRef (sig (MethodName:Id, —), Class:ClassType), Str: String :: Class, [ParamStr: String :: — ]

ifAux (Str < String ParamStr, -1, ifAux (Str == String ParamStr, 0, 1))

REQUIRES Class =_{K} class String ∩_{Bool} Id2String (MethodName) == String "compareTo"

A.13.4 Array clone

RULE ARRAY-CLONE

invokeImpl (methodRef (sig (MethodName:Id, —), Class:ClassType), arrayRef (array0f ElemT, Loc:Int, Len:Int) :: CompileT:RefType, [‘KList ]

arrayCopy (arrayRef (array0f ElemT:Type, Loc:Int, Len:Int) :: CompileT,
 allocAndInitArray (ElemT, [Len :: int ], ⊥ :: ElemT))

REQUIRES Class =_{K} class ArrayImpl ∩_{Bool} Id2String (MethodName) == String "clone"

Construct to copy content of an array to another after copy, the term is rewritten into the destination array.

SYNTAX KItem ::= arrayCopy (TypedVal, K) [strict(2)]

RULE

arrayCopy (arrayRef (T:Type, L1: Int, Len: Int) :: —, arrayRef (T, L2: Int, Len) :: —)

storeCopy (L1, L2, Len) ↼ arrayRef (T, L2: Int, Len)

Copy a sequence of store values from one place to another.

SYNTAX KItem ::= storeCopy (Int, Int, Int)

RULE storeCopy

storeCopy (L1: Int, L2: Int, Len: Int)

storeCopy (L1 + _ {Int} 1, L2 + _ {Int} 1, Len − _ {Int} 1)

store

REQUIRES Len > _ {Int} 0
A.13.5 Class literal operator — A.class

We need to synchronize access to class literals to avoid instantiation of the same .class by multiple threads. This is done by the term temp in <classLiteralsMap>. We cannot use 'Synchronized because it required an objectRef as argument, and we might not have any objects instantiated at the moment.

**Syntax**

\[ KItem ::= \text{saveClassLiteral}(Type, K) ~\text{[strict(2)]} \]

**Rule saveClassLiteral**

\[
\begin{align*}
\text{saveClassLiteral} & \quad (T:Type, TV:TypedVal) \quad \rightarrow \quad \kappa \quad (T \rightarrow temp) \\
\end{align*}
\]

\[
\begin{align*}
\text{classLiteralsMap} & \quad \rightarrow \quad TV \\
\end{align*}
\]

**Rule Lit-Class**

\[
\begin{align*}
\text{Lit('Class(T:Type))} \quad \rightarrow \quad TV \\
\end{align*}
\]

Used inside <classLiteralsMap> only for synchronization purposes

**Syntax**

\[ KItem ::= \text{temp} \]

A.14 Module API-THREADS

Java API related to threads and locks. Just the core part.
A.14.1 Method Thread.start()

RULE THREAD-START
\[
\text{Thread.start()} \quad \text{Thread.start()}
\]
\[
\begin{align*}
\text{invokeImpl ( methodRef ( sig ( MethodName:Id, —), Class:ClassType),}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\begin{align*}
\text{objectRef (OL:Int, —) :: —, [ TId:Int :: — ]}}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\text{nothing :: void}
\hspace{1cm} \quad k \quad \text{thread}
\]
\[
\quad \text{‘Bag}
\]
\[
\quad \text{Field (cast ( class String2Id ("java.lang.Thread"), typedLookup (OL))}
\]
\[
\quad \text{k \quad \text{thread}}
\]
\[
\quad \text{)} \text{String2Id ("runnable") \text{. String2Id ("run")(TypedVals) ;}
\]
\[
\quad \text{k \quad \text{thread}}
\]
\[
\quad \text{String2Id ("run") (TypedVals)}
\]
\[
\quad \text{k \quad \text{thread}}
\]
\[
\quad \langle\langle\langle
\quad \text{TId \_tid}
\]
\[
\quad \text{} \quad \text{\rangle\rangle\rangle}\;
\]
\[
\quad \text{tid}
\]
\[
\quad \text{k \quad \text{thread}}
\]
\[
\text{REQUIRES Class =}_K \text{class String2Id ("java.lang.Thread") } \wedge \text{Id2String (MethodName) } \quad \text{==String "startImpl"}
\]

A.14.2 Synchronized statement

RULE SYNCHRONIZED-FIRST-TIME
\[
\begin{align*}
\text{synchronized ( objectRef (OL:Int, —) :: —) Block:K}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\begin{align*}
\text{try Block’CatchClauses finally releaseLock (OLd)}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\text{busy}
\]
\[
\text{Busy : Set}
\]
\[
\text{OLd}
\]
\[
\text{REQUIRES } \lnot \text{Bool ( OLd in Busy )}
\]
\[
\quad \text{[transition-threading, transition-sync]}
\]

RULE SYNCHRONIZED-NESTED
\[
\begin{align*}
\text{synchronized ( objectRef (OL:Int, —) :: —) Block:K}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\begin{align*}
\text{try Block’CatchClauses finally releaseLock (OLd)}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\begin{align*}
\text{OLd } \leftrightarrow \text{ Level:Int}
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\text{Level } + \text{Int 1}
\]
\[
\text{[transition-threading, transition-sync]}
\]

RULE SYNCHRONIZED-ON-NULL
\[
\begin{align*}
\text{synchronized ( null :: —) }
\hspace{1cm} \quad k \quad \text{thread}
\end{align*}
\]
\[
\text{throw new class NullPointerException ( null :: class String ) ;}
\]

Release one level of lock for the given object.

SYNTAX \quad KItem ::= releaseLock ( Int )
A.14.3 Thread.join()

RULE THREAD-JOIN

\[
\begin{align*}
\text{invokeImpl} & \left( \text{methodRef} \left( \text{sig} \left( \text{MethodName:Id}, \ldots \right), \text{Class:ClassType} \right), \ldots, \\
& \left[ \text{TId:Int} :: \ldots \right] \right) \\
& \text{nothing :: void} \\
& \langle \text{false} \rangle \text{interrupted} \langle \ldots \text{TId} \ldots \rangle \text{terminated} \\
\text{REQUIRES} & \text{Class} =_{K} \text{class String2Id ("java.lang.Thread") } \land \text{Bool Id2String (MethodName)} \\
& \text{==String "joinImpl"} \\
& \text{[transition-threading, transition-sync]} \\
\end{align*}
\]

RULE THREAD-JOIN-INTERRUPTED

\[
\begin{align*}
\text{invokeImpl} & \left( \text{methodRef} \left( \text{sig} \left( \text{MethodName:Id}, \ldots \right), \text{Class:ClassType} \right), \ldots, \\
& \left[ \text{TId:Int} :: \ldots \right] \right) \\
& \text{throw new class String2Id ("java.lang.Interrupted exception") (null :: class String)}; \\
& \langle \text{true} \rangle \text{interrupted} \langle \text{false} \rangle \\
\text{REQUIRES} & \text{Class} =_{K} \text{class String2Id ("java.lang.Thread") } \land \text{Bool Id2String (MethodName)} \\
& \text{==String "joinImpl"} \\
& \text{[transition-threading, transition-sync]} \\
\end{align*}
\]

A.14.4 Methods wait() and notify() — core rules

When object.wait() is called the method call expression is replaced by waitImpl() — an auxiliary function that is used later to exit from the waiting state. The id of the current thread (TId) has to be registered in the set inside \langle \text{waitingThreads} \rangle. The cell \langle \text{holds} \rangle attached to each thread stores the number of times the current thread acquired the lock on each object. Here we use it to make sure that the current thread acquired the lock at least once (see the side condition). Otherwise calling the method wait() is illegal and we have to throw an exception, according to Java API. The exceptional case is covered by other rules. Another cell matched here is \langle \text{busy} \rangle. It stores the set of objects that serve as synchronization monitors — arguments of blocks synchronized. When an object enters the waiting state it have to release the ownership of this
monitor, this is reflected by deleting the entry OId from the set. In this rule the cell \(\langle \rangle\) is used solely for clarity, to separate the cells that are attached to a thread from global cells. In fact, if we would delete \(\langle \rangle\) here and keep just what is inside, the semantics of the rule would not change. This is because configuration abstraction mechanism of \(\mathbb{K}\) would infer the surrounding context for each of \(\langle \rangle\) and \(\langle \rangle\). It would know to match them under the same \(\langle \rangle\), even if there are multiple \(\langle \rangle\) cells.

**RULE OBJECT-WAIT**

\[
\text{objectRef (OId:Int, —) :: — . methodRef (sig (Method:Id, —), Class:ClassType)(TypedVals)}
\]

\[
\text{waitImpl (OId)}
\]

\[
\text{TId:Int • OId \rightarrow HoldLevel:Int \rightarrow \text{thread}}
\]

\[
\text{OId • \text{busy}}
\]

\[
\text{TId \rightarrow OId • \text{waitingThreads}}
\]

**REQUIRES**

\[
\text{Class =}_K \text{class Object} \land \text{Bool Id2String (Method) == String "wait" \land Bool HoldLevel} \geq \text{Int} 1
\]

When another thread calls notify() we have to make sure that the thread holds the monitor on the target object (the side condition). Otherwise we have to throw an exception. The actual logic of notify() is delegated to notifyImpl(), in order to avoid duplication. The construct notifyImpl() requires two rules for two cases — the case when there is at least one thread waiting on the target object, and the case when there is no one.

**RULE OBJECT-NOTIFY**

\[
\text{objectRef (OId:Int, —) :: — . methodRef (sig (Method:Id, —), Class:ClassType)(TypedVals)}
\]

\[
\text{notifyImpl (OId)}
\]

\[
\text{OId \rightarrow HoldLevel:Int \rightarrow \text{thread}}
\]

**REQUIRES**

\[
\text{Class =}_K \text{class Object} \land \text{Bool Id2String (Method) == String "notify" \land Bool HoldLevel} \geq \text{Int} 1
\]

Here we present the rule for the first case. If there is a thread waiting on the current object, then the object identifier OId will be present among the map values of \(\langle \rangle\). By deleting the whole entry associated to that value we enable the waiting thread to proceed. If there is no thread waiting for this object then the term notifyImpl() is simply consumed.

**RULE NOTIFYIMPL-SOMEONE-WAITING**

\[
\text{notifyImpl (OId:Int)}
\]

\[
\text{nothing :: void}
\]

**[transition-threading]**

At this stage the rule for waitImpl() could match. The rule checks in its side conditions that the current thread id TId is not among the waiting threads anymore. It also checks that the target object, represented by OId is not busy. This is required because the thread exiting from waiting state have to reacquire the monitor

184
on the target object. Finally, the rule have to make sure that the thread was not interrupted while it was waiting. Otherwise another rule will match and will throw the appropriate exception.

**RULE WAITIMPL-MAIN**

\[
\begin{align*}
\text{waitImpl (OId: Int)} & \rightarrow k \langle TId: Int \rangle_{tid} \langle  \text{Busy: Set (OId)} \rightarrow \text{busy (false) } \rangle_{interrupted} \\
\langle WT: Map \rangle_{\text{waitingThreads}} & \text{requires (¬ Bool TId in keys (WT))} \land (¬ Bool OId in Busy) \\
\end{align*}
\]

**A.14.5 Object.wait() — additional**

**RULE OBJECT-WAIT-NOTIFY-NOTIFYALL-WITHOUT-SYNC**

\[
\begin{align*}
\text{objectRef (OL: Int, —)} & \rightarrow \langle \text{methodRef (sig (MethodName:Id, —), Class:ClassType)} (\langle \text{TypedVals} \rangle \rightarrow \rangle) \rightarrow k \\
\langle \text{Holds: Map} \rangle_{\text{holds}} & \text{requires Class} = K \text{class Object} \land (\text{Id2String (MethodName)} == \text{String "wait")} \\
\forall \text{Bool Id2String (MethodName)} == \text{String "notify"} \lor \text{Id2String (MethodName)} == \text{String "notifyAll"} \land (\text{¬ Bool OL in keys (Holds)}) \\
\end{align*}
\]

Second part of a wait. Waiting was already registered to <waitingThreads>. When the record in <waitingThreads> will be deleted, waiting itself shall end.

**SYNTAX**

\[ KItem ::= \text{waitImpl (Int) } \]

**RULE WAITIMPL-INTERRUPTED**

\[
\begin{align*}
\text{waitImpl (OL: Int)} & \rightarrow k \langle \text{null :: class String } \rangle; \\
\langle TId: Int \rangle_{tid} \langle  \text{Busy: Set (OId)} \rightarrow \text{busy (false) } \rangle_{interrupted} & \text{requires (¬ Bool OL in Busy) } \\
\end{align*}
\]

Second part of a wait. Waiting was already registered to <waitingThreads>. When the record in <waitingThreads> will be deleted, waiting itself shall end.

**SYNTAX**

\[ KItem ::= \text{notifyImpl (Int) } \]

**A.14.6 Object.notify(), Object.notifyAll() — additional**

Implementation of Object.notify(), extracted here to avoid rule superfluousness

**SYNTAX**

\[ KItem ::= \text{notifyImpl (Int) } \]
rule notifyImpl-someone-waiting

\[ \text{notifyImpl} \left( \text{OL:} \text{Int} \right) \quad \cdot \quad \text{nothing} \mapsto \text{void} \quad \cdot \quad \text{waitingThreads} \]

[transition-threading]

rule notifyImpl-no-one-waiting

\[ \text{notifyImpl} \left( \text{OL:} \text{Int} \right) \quad \cdot \quad \text{nothing} \mapsto \text{void} \quad \cdot \quad \text{waitingThreads} \]

requires \( \neg \text{Bool} \text{OL} \text{in values} \left( \text{WT} \right) \)

[transition-threading]

rule object-notifyAll

\[ \text{objectRef} \left( \text{OL:} \text{Int}, \text{--} \right) \mapsto \text{methodRef} \left( \text{sig} \left( \text{MethodName:} \text{Id}, \text{--} \right), \text{Class:} \text{ClassType} \right) \left( \text{TypedVals} \right) \text{objectNotifyAllImpl} \left( \text{OL} \right) \]

\[ \left( - \text{OL} \mapsto \text{HoldLevel:} \text{Int} - \right) \text{holds} \]

requires \( \text{Class} = \_\text{K} \text{class Object} \wedge \text{Bool Id2String} \left( \text{MethodName} \right) = \text{String} \text{"notifyAll"} \wedge \text{Bool HoldLevel} \geq \text{Int} \text{1} \)

Implementation of \text{Object.notifyAll()}, extracted here to avoid rule superfluosity

SYNTAX \( K\text{Item} ::= \text{objectNotifyAllImpl} \left( \text{Int} \right) \)

rule object-notifyAllImpl-someone-waiting

\[ \text{objectNotifyAllImpl} \left( \text{OL:} \text{Int} \right) \mapsto \text{nothing} \mapsto \text{void} \quad \cdot \quad \text{waitingThreads} \]

[transition-threading]

rule object-notifyAllImpl-no-one-waiting

\[ \text{objectNotifyAllImpl} \left( \text{OL:} \text{Int} \right) \quad \cdot \quad \text{nothing} \mapsto \text{void} \quad \cdot \quad \text{waitingThreads} \]

requires \( \neg \text{Bool} \text{OL} \text{in values} \left( \text{WT} \right) \)

[transition-threading]
A.14.7 Thread.interrupt()

RULE THREAD-INTERRUPT

\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType),}
\]
\[
\text{objectRef (OL:Int, —) :: —, [ TId:Int :: — ] )}
\]
\[
\text{nothing :: void}
\]
\[
\langle \text{TId} \rangle_{\text{tid}} \langle \text{true} \rangle_{\text{interrupted}} \langle \text{thread} \rangle
\]

REQUIRES Class =_K class String2Id ("java.lang.Thread") \& Bool Id2String (MethodName)

==String "interruptImpl"

[transition-threading, transition-sync]

RULE THREAD-INTERRUPT-ENDED-THREAD

\[
\text{invokeImpl ( methodRef ( sig (MethodName:Id, —), Class:ClassType), objectRef (}
\]
\[
\text{OL:Int, —) :: —, [ TId:Int :: — ] )}
\]
\[
\text{nothing :: void}
\]
\[
\langle - \text{TId} \rangle_{\text{terminated}}
\]

REQUIRES Class =_K class String2Id ("java.lang.Thread") \& Bool Id2String (MethodName)

==String "interruptImpl"

[transition-threading, transition-sync]

A.14.8 Thread termination

RULE ThreadTermination

\[
\langle \langle \mathcal{K} \rangle_{\text{k}} \langle \mathcal{H}:\text{Map} \rangle_{\text{holds}} \langle \text{TId:Int} \rangle_{\text{tid}} \rangle_{\text{thread}}
\]
\[
\langle \text{Busy:Set} \rangle_{\text{busy}}
\]
\[
\langle \text{Bag} \rangle_{\text{Bag}}
\]
\[
\langle \text{terminated} \rangle_{\text{terminated}}
\]
\[
\langle \text{ExecutionPhase} \rangle_{\text{globalPhase}}
\]
\[
\langle \text{false} \rangle_{\text{dissolveEmptyK}}
\]

A.14.9 Debug aids

Dissolve all cells except <out> for the purpose of model checking. We will avoid duplicates caused by the same <out> but different order of content in other cells.

RULE DissolveAllExceptOut

\[
\langle \langle \text{Out:List} \rangle_{\text{out}} \langle \text{Bag} \rangle_{\text{threads}} \langle \text{ExecutionPhase} \rangle_{\text{globalPhase}} \langle \text{true} \rangle_{\text{dissolveAllExceptOut}} \rangle_{\text{T}}
\]
\[
\langle \langle \text{Out} \rangle_{\text{out}} \rangle_{\text{T}}
\]
A.15 Module UNFOLDING

The Unfolding phase of the execution semantics, opposite to the folding phase of the preprocessing semantics. In this phase the preprocessed program stored in <program> cell is distributed into various cells inside <classes>.

At the beginning of execution semantics <k> contains the term unfoldingPhase, <program> is non-empty and the global phase is UnfoldingPhase.

SYNTAX  

\[ KItem ::= \text{unfoldingPhase} \]

In the initial configuration the cell <threads> is empty, global phase is UnfoldingPhase, <program> is non-empty.

RULE UnfoldingPhase-start

\[ \begin{array}{l}
\text{thread} \leftarrow \text{UnfoldingPhase} \text{globalPhase} \leftarrow \text{program} \\
\end{array} \]

RULE Unfolding-class-unfold-top-level

\[ \begin{array}{l}
\text{class} \leftarrow \text{EnclosingClass} \leftarrow \text{EnclosingClass} \leftarrow \text{getDecFromDec} \leftarrow \text{KList} \leftarrow \text{KList} \leftarrow \text{class} \\
\end{array} \]

RULE Unfolding-class-unfold-inner

\[ \begin{array}{l}
\text{class} \leftarrow \text{EnclosingClass} \leftarrow \text{EnclosingClass} \leftarrow \text{getDecFromKDec} \leftarrow \text{KList} \leftarrow \text{KList} \leftarrow \text{class} \\
\end{array} \]

REQUIRES  \[ \text{ClassDecKL} = KLabel \text{ClassDec} \lor \text{Bool} \text{ClassDecKL} = KLabel \text{InterfaceDec} \]

SYNTAX  

\[ KItem ::= \text{getDecFromDec} \leftarrow (K) \leftarrow \text{function} \]
RULE

getClassFromClassDec (—:KLabel('ClassDecHead(—:K, Class:ClassType, —:KList), —))

Class

RULE UNFOLDING-CLASS-INHERIT-METHODS-NOT-OBJECT

(unfoldingPhase )

{Class:ClassType }classType

BaseMethods

'Map

methods

FoldedCPhase

UnfoldingStartedCPhase

classPhase

{<BaseClass }classType {<BaseMethods:Map }methods {UnfoldedCPhase }classPhase """class

RULE UNFOLDING-CLASS-INHERIT-METHODS-OBJECT-AND-INTERFACES

(unfoldingPhase )

{KL:KLabel('ClassDecHead(—:K, Class, —:K, 'Some('Super Dec(BaseClass:ClassType)), —:K), —)}folded

FoldedCPhase

UnfoldingStartedCPhase

classPhase

RULE UNFOLDING-CLASS-MEMBERS-EXCEPT-METHODS

(unfoldingPhase )

{Class:ClassType }classType

getClassMetaType ( KL)

classMetaType

BaseClass:ClassType

extends

[ InstanceFields:KList ]

instanceFields

[ StaticFields:KList ]

staticFields

[ StaticInit:KList ]

staticInit

KL:KLabel('ClassDecHead( AccessMode:AccessMode, Class, 'None(KList), 'Some('Super Dec(BaseClass:ClassType)), 'Some('ImplementsDec([ — 1 ])), 'ClassBody([ StaticFields:KList, 'StaticInit('Block(StaticInit:KList)'), InstanceFields:KList ])

'KList

folded

UnfoldingStartedCPhase )classPhase

REQUIRES ¬Bool.containsMethodDecs ( [ InstanceFields ] ) =_K true

SYNTAX  KItem ::= getClassMetaType ( KLabel ) [function]
RULE

getClassMetaType ('ClassDec)

    class

RULE

getClassMetaType ('InterfaceDec)

    interface

SYNTAX  \textit{KItem ::= containsMethodDecs (KListWrap) [function]}

RULE

containsMethodDecs ( [ —, 'MethodDec(—), — ])

    true

RULE UNFOLDING-IMPLEMENTSDEC

\langle \text{unfoldingPhase} \rangle_k \langle \text{Class:ClassType} \rangle_{\text{classList}} \langle \ldots \text{Set} \rangle_{\text{BaseInterface:ClassType}} \langle \ldots \text{implTrans} \rangle_{\text{KList}} \langle \ldots \text{folded} \rangle_{\text{UnfoldingStartedCPhase}}_{\text{classPhase}}

RULE UNFOLDING-METHODDEC

\langle \text{unfoldingPhase} \rangle_k \langle \text{Class:ClassType} \rangle_{\text{classList}} \langle \text{Bag} \rangle \langle \text{methodSignature} \rangle \langle \text{methodParams} \rangle \langle \text{methodBody} \rangle \langle \text{methodAccessMode} \rangle \langle \text{methodContextType} \rangle \langle \text{Env:Map} \rangle \langle \text{methods} \rangle \langle \text{folded} \rangle_{\text{UnfoldingStartedCPhase}}_{\text{classPhase}}
A.16 Module TO-STRING

Converts all possible Java values into String. Used mostly for printing and String + operator. Conversion for other value forms is defined in other modules.

**SYNTAX**

```
Exp ::= toString ( K ) [strict]
```

**RULE**

```
toString ( Str: String :: — )
```

```
Str
```

**RULE**

```
toString (I: Int :: char )
```

```
chrChar (I)
```
RULE

\[
\text{toString} \ (I: \text{Int} :: T: \text{Type}) \\
\quad \text{Int2String} \ (I)
\]

REQUIRES \( T \neq \text{char} \)

RULE

\[
\text{toString} \ (F: \text{Float} :: \_)
\quad \text{Float2String} \ (F)
\]

RULE

\[
\text{toString} \ (\text{true} :: \_)
\quad \text{"true"}
\]

RULE

\[
\text{toString} \ (\text{false} :: \_)
\quad \text{"false"}
\]

RULE

\[
\text{toString} \ (\text{null} :: \_)
\quad \text{"null"}
\]

\[
\text{toString}(\text{arrayOf } T)
\]

RULE

\[
\text{toString} \ (\text{arrayOf byte})
\quad \text{"[B"}
\]

RULE

\[
\text{toString} \ (\text{arrayOf short})
\quad \text{"[S"}
\]

RULE

\[
\text{toString} \ (\text{arrayOf int})
\quad \text{"[I"}
\]

RULE

\[
\text{toString} \ (\text{arrayOf long})
\quad \text{"[J"}
\]
RULE

\[
\text{toString( } \text{arrayOf char } \text{)}
\]

"[C"

RULE

\[
\text{toString( } \text{arrayOf bool } \text{)}
\]

"[Z"

RULE

\[
\text{toString( } \text{arrayOf class Class:Id} \text{)}
\]

"[L" + { toString( class Class ) + "." }

RULE

\[
\text{toString( } \text{arrayOf arrayOf T:Type} \text{)}
\]

"[" + toString( arrayOf T )

RULE toString-ClassType-TopLevelWithPackage

\[
\underbrace{\text{toString( class ClassId:Id) }}_{\text{Id2String( ClassId) }} \text{""} \text{class ClassId } \text{classType } \langle \kappa \rangle \text{enclosingClass}
\]

REQUIRES retainHead( Id2String( ClassId ), 1) 

RULE toString-ClassType-TopLevelWithDefaultPackage

\[
\underbrace{\text{toString( class ClassId:Id) }}_{\text{trimHead( Id2String( ClassId), 1) }} \text{""} \text{class ClassId } \text{classType } \langle \kappa \rangle \text{enclosingClass}
\]

REQUIRES retainHead( Id2String( ClassId ), 1) ==String "."

RULE toString-ClassType-Inner

\[
\underbrace{\text{toString( class ClassId:Id) }}_{\text{toString( class OuterClassId) } + \text{\""} + \text{String2Id ( getSimpleName( "" ) class ClassId )))}} \text{class ClassId } \text{classType } \langle \kappa \rangle \text{enclosingClass}
\]

toString for objectRef

RULE

\[
\text{toString( objectRef( OId:Int, LowestClass:ClassType) :: T:Type) }
\]

objectRef( OId, LowestClass ) :: T . (( String2Id ("toString") )\langle\text{TypedVals}\rangle}
A.16.1 Debug helper functions

**SYNTAX**  
\[ KItem ::= debugPrint ( K ) \]

**RULE**
\[
\begin{array}{l}
\text{debugPrint} ( K : K ) \\
\rightarrow \ K \rightarrow \ \text{List} \\
\rightarrow \ K
\end{array}
\]

**SYNTAX**  
\[ KItem ::= debugString ( K ) \ [\text{function}] \]
\[ | \text{debugStringList} ( KListWrap ) \ [\text{function}] \]
\[ | \text{Bool2String} ( \text{Bool} ) \ [\text{function}] \]

**RULE**
\[
\text{debugString} ( KI : KItem \rightarrow KI2 : KItem \rightarrow K : K )
\]
\[
\text{debugString} ( KI : KItem ) + \text{String} " \rightarrow " + \text{String} \text{debugString} ( KI2 : KItem \rightarrow K : K )
\]

**RULE**
\[
\text{debugString} ( X : \text{Id} )
\]
\[
\text{Id2String} ( X )
\]

**RULE**
\[
\text{debugString} ( I : \text{Int} )
\]
\[
\text{Int2String} ( I )
\]

**RULE**
\[
\text{debugString} ( B : \text{Bool} )
\]
\[
\text{Bool2String} ( B )
\]

**RULE**
\[
\text{debugString} ( Fl : \text{Float} )
\]
\[
\text{Float2String} ( Fl )
\]

**RULE**
\[
\text{debugString} ( Str : \text{String} )
\]
\[
\text{Str}
\]

**RULE**
\[
\text{debugString} \ ( \text{types} ( Ks : \text{KList} ) )
\]
\[
"\text{types}(" + \text{String} \text{debugStringList} ( [ \text{Ks} ] ) + \text{String }")"
\]
RULE 

\[
\text{debugString ( [ Ks:KList ] )}
\]

"[" + String debugStringList ( [ Ks ] ) + String "]"

RULE 

\[
\text{debugString (KL:KLabel(Ks:KList))}
\]

KLabel2String (KL) + String "(" + String debugStringList ( [ Ks ] ) + String ")"

REQUIRES \( \neg \text{Bool isDebugStringLeaf (KL(Ks))} =_K \text{true} \) \land \text{Bool isString(KLabel2String (KL))} =_K \text{true} \land \text{Bool Ks} /=_K \text{List 'KList}

RULE 

\[
\text{debugString (KL:KLabel(KList))}
\]

KLabel2String (KL)

REQUIRES \( \neg \text{Bool isDebugStringLeaf (KL(KList))} =_K \text{true} \) \land \text{Bool isString(KLabel2String (KL))} =_K \text{true}

RULE 

\[
\text{debugString (KL:KLabel(Ks:KList))}
\]

"???"

REQUIRES \( \neg \text{Bool isDebugStringLeaf (KL(Ks))} =_K \text{true} \) \land \text{Bool \neg Bool isString(KLabel2String (KL))} =_K \text{true}

RULE 

\[
\text{debugString ('K)}
\]

"

RULE 

\[
\text{debugStringList ( [ K1:K, Ks:KList ] )}
\]

debugString (K1:K) + String ". " + String debugStringList ( [ Ks:KList ] )

REQUIRES Ks =/= KList 'KList

RULE 

\[
\text{debugStringList ( [ K1:K ] )}
\]

debugString (K1)

RULE 

\[
\text{debugStringList ( [ 'KList ] )}
\]

""
RULE

Bool2String (true)

"true"

RULE

Bool2String (false)

"false"

SYNTAX  \[KItem ::= \text{isDebugStringLeaf} (K)\]

RULE

isDebugStringLeaf (\text{—:Id})

true

RULE

isDebugStringLeaf (\text{—:Int})

true

RULE

isDebugStringLeaf (\text{—:Bool})

true

RULE

isDebugStringLeaf (\text{—:Float})

true

RULE

isDebugStringLeaf (\text{—:String})

true

RULE

isDebugStringLeaf (\text{—:Types})

true

RULE

isDebugStringLeaf (\text{[ — ]})

true
A.17   Module SYNTAX-CONVERSIONS

Defining the syntax for key portions of the semantics, and converting AST terms back into their syntactic form. This whole module is auxiliary, its purpose is to allow defining some portions of the semantics syntactically. It will disappear once the main syntax of K-Java will be used instead.

A.17.1    Method parameter

**SYNTAX** \( KItem ::= \text{toParams}\ (KListWrap, Params) \) [function]

**RULE**
\[
\begin{align*}
toParams &\ ( [ KLParams:KList, 'Param(—, K:K, X:Id) ], \text{Params:Params} ) \\
&
\text{toParams} ( [ KLParams ], ( KX, \text{Params} ) )
\end{align*}
\]

**RULE**
\[
\begin{align*}
toParams &\ ( [ KList ], \text{Params:Params} ) \\
&
\text{Params}
\end{align*}
\]
Required for getTypes()

**RULE**
\[
\begin{align*}
typeOf &\ ('ParamImpl(T:Type, —:Id)) \\
&
T
\end{align*}
\]

A.17.2    Method invocation

**RULE**
\[
\begin{align*}
'\text{Invoke}' &\ ('Method('MethodName(Qual:K, Name:Id)), ArgList:KListWrap) \\
&
\text{Qual} \ . \ Name ( \text{toExps} ( \text{ArgList} ) )
\end{align*}
\]
[structural]

A.17.3    Local variable declaration

**SYNTAX** \( LocalVarDeclStmt ::= Type \ Id \\)
RULE

\[
\begin{align*}
\text{'LocalVarDec} & \quad (\rightarrow K, T:Type, \text{ [ 'VarDec(X:Id) ] }) \\
& \quad TX;
\end{align*}
\]

[structural]

A.17.4 Cast

SYNTAX \[ Exp ::= (( Type ) K ) \quad \text{[klabel('CastImpl)]} \]

RULE

\[
( ( T : Type ) K : K ) \\
\quad \text{cast} ( T, K )
\]

A.17.5 Identifier (name) expression

SYNTAX \[ Exp ::= Id \]

RULE

\[
X : Id \\
\quad \text{'ExprName}(X)
\]

RULE

\[
\text{lvalue} ( X : Id ) \\
\quad \text{lvalue} ( \text{'ExprName}(X) )
\]

RULE

\[
\text{'NewInstance}(\text{'None(KList)}, \text{Class:ClassType}, \text{ [ Args:KList ]}, \text{'None(KList)}) \\
\quad \text{new Class( toExps ( [ Args ] ))}
\]

RULE

\[
\text{'QNewInstance}(\text{Qual:Exp}, \rightarrow K, \text{ClassOrName:ClassOrName}, \rightarrow K, \text{ [ Args:KList ]}, \rightarrow K) \\
\quad \text{Qual . new ClassOrName( toExps ( [ Args ] ))}
\]

Unpack 'NewInstance back into 'QNewInstance. The difference between desugaring and unpacking is that unpacking is an artificial procedure required to separate elaboration from execution. In the elaboration phase there is a opposite packing rule that creates this construct.
**RULE** \textsc{NewInstance-to-QNewInstance-unpack}

\[
\begin{align*}
\text{new Class:ClassType} & (\text{ArgExps:Exps}) \\
\kappa & . \text{new Class:ClassType} (\text{ArgExps:Exps})
\end{align*}
\]

### A.17.6 Syntactic lists

A list of expressions, usually a list of arguments of a method or constructor.

**SYNTAX** \( KItem ::= \text{toExps} ( KLListWrap ) \) [function]

\[
| \text{toExps} ( KLListWrap, \text{Exps} ) \) [function]
\]

**RULE**

\[
\text{toExps} ( [ \text{Args:KLList} ] ) \\
\text{toExps} ( [ \text{Args} ], \text{'Exps} )
\]

**RULE**

\[
\text{toExps} ( [ \text{Args:KLList, Arg:K} ], \text{Es:K} ) \\
\text{toExps} ( [ \text{Args} ], ( \text{Arg, Es} ) )
\]

**RULE**

\[
\text{toExps} ( [ \text{'KLList} ], \text{Es:K} ) \\
\text{Es}
\]

**SYNTAX** \( KItem ::= \text{toKLListWrap} ( \text{Exps} ) \) [function]

\[
| \text{toKLListWrap} ( KLListWrap, \text{Exps} ) \) [function]
\]

**RULE**

\[
\text{toKLListWrap} ( \text{Args:Exps} ) \\
\text{toKLListWrap} ( [ \text{'KLList} ], \text{Args} )
\]

**RULE**

\[
\text{toKLListWrap} ( [ \text{Args:KList}, ( \text{Arg:Exp, Es:Exps} ) ] ) \\
\text{toKLListWrap} ( [ \text{Args, Arg} ], ( \text{Es} ) )
\]

**RULE**

\[
\text{toKLListWrap} ( [ \text{Args:KList} ], \text{'Exps} ) \\
[ \text{Args} ]
\]

**SYNTAX** \( \text{Exps ::= TypedVals} \)
SYNTAX  \( \text{TypedVals} ::= \text{List}\{\text{TypedVal}, \text{""}, \text{"}\} \)

Converts a TypedVals term to Types. Uses typeOf underneath.

SYNTAX  \( K\text{Item} ::= \text{getTypes } ( \text{TypedVals} ) \ [\text{function}] \)
          | \( \text{getTypes } ( \text{Types}, \text{TypedVals} ) \ [\text{function}] \)

RULE

\[
\begin{align*}
\text{getTypes}(\text{TVs}:\text{TypedVals}) \\
\text{getTypes}(\text{types}(\text{"KL}\text{ist}), \text{TVs})
\end{align*}
\]

RULE

\[
\begin{align*}
\text{getTypes}(\text{types}(\text{Ts}:\text{KList}), (--- :: \text{T:Type, TVs}:\text{TypedVals})) \\
\text{getTypes}(\text{types}(\text{Ts}, \text{T}), \text{TVs})
\end{align*}
\]

RULE

\[
\begin{align*}
\text{getTypes}(\text{types}(\text{Ts}:\text{KList}), \text{"TypedVals}) \\
\text{types}(\text{Ts})
\end{align*}
\]

SYNTAX  \( K\text{Result} ::= \text{ClassTypes} \)

SYNTAX  \( \text{ClassTypes} ::= \text{List}\{\text{ClassType}, \text{""}, \text{"}\} \)

Try/catch

RULE

\[
\begin{align*}
\text{′Try}(\text{Trys}:K, \ [\text{Catches}:\text{KList}] \\
\text{toCatchClauses}(\ [\text{Catches}]) \)
\end{align*}
\]

[anywhere]

SYNTAX  \( \text{CatchClauses ::= toCatchClauses } ( \text{KList\Wrap} ) \ [\text{function}] \)
          | \( \text{toCatchClauses } ( \text{KList\Wrap}, \text{CatchClauses} ) \ [\text{function}] \)

RULE

\[
\begin{align*}
\text{toCatchClauses}(\ [\text{KL\CatchClauses}:\text{KList} ] ) \\
\text{toCatchClauses}(\ [\text{KL\CatchClauses}], \text{"CatchClauses})
\end{align*}
\]

RULE

\[
\begin{align*}
\text{toCatchClauses}(\ [\text{KL\CatchClauses}:\text{KList}, \text{C:CatchClause}], \text{CatchClauses}:\text{CatchClauses}) \\
\text{toCatchClauses}(\ [\text{KL\CatchClauses}], (\text{C CatchClauses}))
\end{align*}
\]
RULE

toCatchClauses ( [ \text{KL}\text{ist} ], \text{CatchClauses}:\text{CatchClauses} )

\text{CatchClauses}

CONTEXT

\text{try} \quad \text{----} : K \square 

SYNTAX  \quad \text{CatchClause} ::= \text{catch} ( K ) K \quad \text{klabel} ( '\text{Catch} ) 

CONTEXT

\quad \text{catch} ( \square ) --

\quad [ \text{result}(\text{ResultOrParam}) ] 

RULE

\quad \text{'Param(\text{----}, K:Type, X:Id)}

\quad K \quad X

\quad [ \text{structural} ] 

RULE

\quad \text{catch} ( \text{KR:ResultOrParam} ) \quad S \quad K

\quad \text{catchImpl} ( \text{KR}, S )

Extended K Result. Represents KLabels that should be treated as KResult during execution phase, but not during elaboration phase.

SYNTAX  \quad \text{ResultOrParam} ::= \text{Param} 

RULE

\quad \text{isResultOrParam(\text{----}:KResult)}

\quad \text{true} 

Internal representation of a preprocessed catch clause

SYNTAX  \quad \text{CatchImpl ::= catchImpl ( K, K )} 

SYNTAX  \quad \text{KResult ::= CatchImpl} 

SYNTAX  \quad \text{CatchClause ::= CatchImpl}
Appendix B

K-Java Common modules

B.1 Module CORE-SORTS

This module contains general-purpose utilities for the whole K-Java, both preprocessing and execution. This is the lowest-level module of the whole semantics, imported by almost all other modules. Because of its high accessibility status, it was somewhat abused, in a sense that I placed here some random stuff for the lack of better place.

B.1.1 Computation phases

The computation Global Phase. See configuration documentation for details.

```
SYNTAX  GlobalPhase ::= ProcTypeNamesPhase |
| ProcCompUnitsPhase |
| ProcClassDecsPhase |
| ProcClassMembersPhase |
| ElaborationPhase |
| FoldingPhase |
| UnfoldingPhase |
| ExecutionPhase |
```

the class phase. See configuration documentation for details.

```
SYNTAX  ClassPhase ::= DiscoveredCPhase |
| StoredCPhase |
| BasesResolvedCPhase |
| DecsProcessedCPhase |
| MembersProcessedCPhase |
| FoldedCPhase |
| UnfoldingStartedCPhase |
| UnfoldedCPhase |
```

The state of a class in relation to static initialization. See static-init.k for more details.

```
SYNTAX  StaticInitStatus ::= StaticUninitialized |
| StaticInitializing (Int) |
| StaticInitialized |
```
B.1.2 Values

SYNTAX  \( \text{RawVal} ::= \text{Int} \)
\| \( \text{Float} \)
\| \( \text{Bool} \)
\| \( \text{RawRefVal} \)
\| \( \text{nothing} \)
\| \( \bot \)

A reference value, in its four forms — as regular object, array, null or String. As mentioned previously, string values have special treatment in this semantics and are implemented as regular strings.

The object closure has the following structure:

- \( \text{OL:Int} \) — the object location
- \( \text{List} \) — the object content, of the form

\[
\text{ListItem(layer(}
\text{Class},
<\text{env}>\text{Env}<\text{env}>,
\text{enclosingObjClosure::T}
\text{))}
\]

\[
\ldots
\]

SYNTAX  \( \text{ObjectRef} ::= \text{objectRef} \ (\text{Int}, \text{ClassType}) \)

The arrayRef has the following structure:

- \( \text{Type} \) — type of the array
- \( \text{Int} \) — location of the first element
- \( \text{Int} \) — array length

SYNTAX  \( \text{ArrayRef} ::= \text{arrayRef} \ (\text{Type}, \text{Int}, \text{Int}) \)

SYNTAX  \( \text{RawRefVal} ::= \text{ObjectRef} \)
\| \( \text{ArrayRef} \)
\| \( \text{String} \)
\| \( \text{NullLiteral} \)

A typed value is a pair of of a raw value and a type. Anywhere during execution we will evaluated typed expressions into typed values, never into raw values alone.

SYNTAX  \( \text{TypedVal} ::= \text{RawVal} :: \text{Type} \)

Since RawVal terms can easily be converted into TypedVal, they are also of sort Exp.
The three most common forms of computation result are Typed value or Type. Those are most common results we will encounter during elaboration. Other result types are more context-specific.

```
SYNTAX   Exp ::= TypedVal
          |  RawVal

KResult ::= TypedVal
          |  Type

KResult ::= 'K
```

## B.1.3 Class and member attributes

A method signature. Required to be KResult by rules in METHOD-INVOKE.

```
SYNTAX   Signature ::= sig ( Id, Types )

SYNTAX   KResult ::= Signature

ContextType ::= static
               |  instance

KResult ::= ContextType

AccessMode ::= Public
             |  Protected
             |  Private
             |  package

KResult ::= AccessMode

MethodMetaType ::= methodMMT
                 |  constructorMMT

ClassMetaType ::= class
                |  interface
```

## B.1.4 Misc definitions

Represents a reference to a method or constructor.

```
SYNTAX   MethodRef ::= methodRef ( Signature, RefType )
                 |  'K
```
SYNTAX  \textit{KResult} ::= \textit{MethodRef}

SYNTAX  \textit{MethodName} ::= \textit{MethodRef}

A list of types. Elements are of sort Type. Used especially to denote argument types of a method.

SYNTAX  \textit{Types} ::= \textit{types} ( \textit{KList} )

SYNTAX  \textit{KResult} ::= \textit{Types}

Location type of each store location — either local variable or field

SYNTAX  \textit{LocMetadata} ::= \textit{Local}
  | \textit{Field}

SYNTAX  \textit{MethodInvokeExp} ::= \textit{K} . \textit{MethodName} ( \textit{Exps} ) \text{[seqstrict(1,3), klabel('MethodImpl)]}

SYNTAX  \textit{ClassInstanceCreationExp} ::= \textit{new} \textit{ClassType} ( \textit{Exps} )
  | \textit{Exp} . \textit{new} \textit{ClassOrName} ( \textit{Exps} ) \text{[seqstrict(1,3)]}

SYNTAX  \textit{ClassOrName} ::= \textit{Id}
  | \textit{ClassType}
  | 'K

A fully qualified class name, or noClass where no valid class exists in the current context.

SYNTAX  \textit{ClassType} ::= \textit{class} \textit{Id}
  | 'K

B.1.5 Random unsorted content, syntax converters

SYNTAX  \textit{Param} ::= \textit{Type} \textit{Id} \text{[klabel('ParamImpl)]}

SYNTAX  \textit{RefType} ::= \textit{ClassType}
  | \textit{nullType}
  | \textit{arrayOf} \textit{Type}

RULE

'ArrayType(T:Type)

\hspace{1cm} arrayOf T
\hspace{1cm} [anywhere]

RULE

'ClassOrInterfaceType(TypeK:K, —)

\hspace{1cm} TypeK
\hspace{1cm} [anywhere]
Convert the AST representation of an Id into a K Id.

RULE
\[
\begin{align*}
\text{'Id(Str: String) } \\
\text{String2Id( Str) } \\
\text{[structural, anywhere]}
\end{align*}
\]

ListWrap and KListWrap

Represent a parser form for a sequence of terms. Is desugared into the pretty form [...]

SYNTAX \(KLabel ::= \text{'ListWrap}\)

RULE
\[
\begin{align*}
\text{'ListWrap(Ks: KList) } \\
\text{[Ks] } \\
\text{[structural, anywhere]}
\end{align*}
\]

SYNTAX \(KListWrap ::= [\ KList\ ]\ [\text{klabel('KListWrap')}]\)

Sequence of terms and of any other statements. The first term is moved to the top of computation.

RULE
\[
\begin{align*}
\text{[ S1: K, Stmts: KList ]} \\
\text{S1 \to [ Stmts ]} \\
\text{[structural]}
\end{align*}
\]

RULE
\[
\begin{align*}
\text{[ 'KList ]} \\
\text{'K} \\
\text{[structural]}
\end{align*}
\]

A wrapper over an arbitrary KList, wrapper being of type KResult.

SYNTAX \(KRLListWrap ::= kr[\ KList\ ]\)

SYNTAX \(KResult ::= KRLListWrap\)

Computation terms produced during elaboration

These auxiliary terms functions that should be threated as expressions in the elaboration phase.

SYNTAX \(Exp ::= AuxTermExp\)
SYNTAX  \[ \text{LHS} ::= \text{AuxTermExp} \]

Wrapper of a statement followed by an expression, to be used in a place where an expression is expected, such as an anonymous class declaration. Is typed with the type of the expression. At runtime is rewritten into the statement, that should evaluate into .K, followed by the expression, that should evaluate to value.

SYNTAX  \[ \text{AuxTermExp} ::= \text{stmtAndExp} \left( \ K, \ K \right) \]

SYNTAX  \[ \text{AuxTermExp} ::= \text{cast} \left( \ Type, \ K \right) \ [\text{strict}] \]

B.2 Module CORE-FUNCTIONS

B.2.1 Core utility functions

Returns the type associated with various K terms. The implementation is scattered among various K files. For a type — the type itself. For a typed expression - the type component. For some raw values - their most common type.

SYNTAX  \[ \text{KItem} ::= \text{typeOf} \left( \ K \right) \ [\text{function}] \]

RULE

\[
\text{typeOf} \left( \ T : \text{Type} \right) \\
\frac{\text{T}}{} 
\]

RULE

\[
\text{typeOf} \left( \ — :: \ T : \text{Type} \right) \\
\frac{\text{T}}{} 
\]

RULE

\[
\text{typeOf} \left( \text{Str} : \text{String} \right) \\
\frac{\text{class String}}{} 
\]

RULE

\[
\text{typeOf} \left( \text{null} \right) \\
\frac{\text{nullType}}{} 
\]

RULE

\[
\text{typeOf} \left( \text{objectRef} \left( \ —, \ \text{Class} : \text{ClassType} \right) \right) \\
\frac{\text{Class}}{} 
\]
RULE
\[
\text{typeof( arrayRef( arrayOf T:Type, --- ) )}
\]
\[
\text{arrayOf T}
\]
Required by \text{getTypes()} in elaboration phase, invoked by \text{lookupMethod()}.  

RULE
\[
\text{typeof( cast( T:Type, --- ) )}
\]
\[
T
\]
Required for \text{getTypes()}

RULE
\[
\text{typeof( 'Param(---:K, T:Type, ---:Id) )}
\]
\[
T
\]
The default value for all types. Used for field initializers.

\text{SYNTAX}

\[KItem ::= \text{default ( Type )} \] [function]

RULE
\[
\text{default ( IntT:IntType )}
\]
\[
0 :: \text{IntT}
\]

RULE
\[
\text{default ( FloatT:FloatType )}
\]
\[
0.0 :: \text{FloatT}
\]

RULE
\[
\text{default ( bool )}
\]
\[
\text{false :: bool}
\]

RULE
\[
\text{default ( RT:RefType )}
\]
\[
\text{null :: RT}
\]
Whenever naked \text{RawVal} reaches the top of computation, it have to be converted into \text{TypedVal}

RULE
\[
\text{Val:RawVal}
\]
\[
\text{toTypedVal ( Val )} \] [structural]

208
Converts a RawVal into a TypedVal, by adding a default type for the given value. Greatly simplifies many rules, because now we can use raw values in the RHS, instead of typed values.

Have to be [function] for LTL

**SYNTAX**  \( KItem ::= \text{toTypedVal} (\text{RawVal}) \) [function]

**RULE**

\[
\text{toTypedVal} (I: Int)
\]

\[
I :: \text{int}
\]

[structural]

**RULE**

\[
\text{toTypedVal} (B: Bool)
\]

\[
B :: \text{bool}
\]

[structural]

**RULE**

\[
\text{toTypedVal} (Str: String)
\]

\[
Str :: \text{class String}
\]

[structural]

**RULE**

\[
\text{toTypedVal} (null)
\]

\[
null :: \text{nullType}
\]

[structural]

**RULE**

\[
\text{toTypedVal} (\text{objectRef} (\text{OL: Int}, \text{Class: ClassType}))
\]

\[
\text{objectRef} (\text{OL: Int}, \text{Class: ClassType}) :: \text{Class}
\]

[structural]

**RULE**

\[
\text{toTypedVal} (\text{arrayRef} (T: Type, L: Int, M: Int))
\]

\[
\text{arrayRef} (T, L, M) :: T
\]

[structural]

Converts a KList of terms to Types. Uses typeOf underneath.

**SYNTAX**  \( KItem ::= \text{getTypes} (\text{KListWrap}) \) [function]
RULE

\[
\text{getTypes ( [ \quad, \quad, \quad ] )}
\]

\text{typeof (K)}

\text{REQUIRES isType(K) } \neq K \text{ true } \land \text{Bool getKLabel (K) } \neq K \text{ 'typeOf '''}

RULE

\[
\text{getTypes ( [ Ts:KList ] )}
\]

\text{types (Ts)}

\text{REQUIRES isKResult(Ts)}

B.2.2 Utilities for general-purpose programming

Generalized equality with strictness, suitable to compare not only TypedVal-s but any K terms.

\text{SYNTAX } KItem ::= eqAux (K, K) [seqstrict]

RULE

\[
\]

\text{KR1 } =_K \text{ KR2}

Alternative version of if, to be used whenever semantics needs an if logic. The original 'If will only be used in the source programs.

\text{SYNTAX } KItem ::= ifAux (K, K, K) [strict(1)]

RULE

\[
\text{ifAux (true :: bool, S:K, ---)}
\]

\text{S}

RULE

\[
\text{ifAux (false :: bool, ---, S:K)}
\]

\text{S}

\text{SYNTAX } KItem ::= andAux (K, K) [strict(1)]

RULE

\[
\text{andAux (true :: bool, E:K)}
\]

\text{E}
RULE
\[
\text{andAux} (\text{false} :: \text{bool}, \rightarrow) \\
\rightarrow \text{false}
\]

SYNTAX \[ KItem ::= \text{orAux} (K, K) \text{[strict(1)]} \]

RULE
\[
\text{orAux} (\text{true} :: \text{bool}, \rightarrow) \\
\rightarrow \text{true}
\]

RULE
\[
\text{orAux} (\text{false} :: \text{bool}, E:K) \\
\rightarrow E
\]

Replaces 'Plus in preprocessing phases

SYNTAX \[ KItem ::= \text{plusAux} (K, K) \text{[strict]} \]

RULE
\[
\text{plusAux} (\text{Str1}:\text{String} :: \rightarrow, \text{Str2}:\text{String} :: \rightarrow) \\
\rightarrow \text{((Str1} + \text{String Str2}) :: \text{class String})
\]

A wrapper for maps. Allows holding maps inside an arbitrary expression where a K term is required. Also used as part of mapUnion operation in PROCESS-CLASS-MEMBERS

SYNTAX \[ KResult ::= \text{mapWrap} (\text{Map}) \]

The union of two maps. Arguments are of the form mapWrap(Map). In this operation, elements of the second map overwrite elements of the first map if they collide.

SYNTAX \[ KItem ::= map\text{Union} (K, K) \text{[strict]} \]

RULE
\[
\text{map\text{Union}} (\text{map\text{Wrap}} (\begin{array}{c}
\text{M1:Map} \\
\hline
\text{M1 [ K2 / K1 ]}
\end{array}), \text{map\text{Wrap}} (\begin{array}{c}
\text{\text{\rightarrow:Map} K1.K \rightarrow K2.K} \\
\hline
\text{'Map}
\end{array}))
\]

RULE
\[
\text{map\text{Union}} (\text{map\text{Wrap}} (\text{M1:Map}), \text{map\text{Wrap}} (\text{\text{\rightarrow:Map}})) \\
\rightarrow \text{map\text{Wrap}} (\text{M1})
\]

Subtracts fron the given map the keys found in the given set
SYNTAX  \[ KItem ::= \text{mapDiff} \ (K, K) \ [\text{strict}] \]

RULE
\[
\text{mapDiff} \ (\text{mapWrap} (\text{mapDiff}\ K \rightarrow - \rightarrow\ Map), \text{setWrap} (\text{mapDiff}\ K \rightarrow - \rightarrow\ Set))
\]

RULE
\[
\text{mapDiff} \ (\text{mapWrap} (\text{MyMap}:\text{Map}), \text{setWrap} (\text{mapDiff}\ K \rightarrow - \rightarrow\ Set))
\]

REQUIRES \( \neg \text{Bool} \) Key in keys (MyMap)

RULE
\[
\text{mapDiff} \ (\text{mapWrap} (\text{MyMap}:\text{Map}), \text{setWrap} (\text{mapDiff}\ K \rightarrow - \rightarrow\ Set))
\]

\[
\text{mapWrap} (\text{MyMap})
\]

SYNTAX  \[ KItem ::= \text{isEmpty} \ (\text{Map}) \ [\text{function}] \]

RULE
\[
\text{isEmpty} \ (\text{mapDiff}\ Map)
\]

\[
\text{true}
\]

A wrapper for sets, similar to the one for maps.

SYNTAX  \[ KResult ::= \text{setWrap} \ (\text{Set}) \]

The union of two sets. Arguments are of the form setWrap(Set).

SYNTAX  \[ KItem ::= \text{setUnion} \ (K, K) \ [\text{strict}] \]

RULE
\[
\text{setUnion} \ (\text{setWrap} (\text{setUnion}\ K \rightarrow \text{Set}), \text{setWrap} (\text{setUnion}\ K \rightarrow \text{Set}))
\]

REQUIRES \( \neg \text{Bool} \) (K in S1)

RULE
\[
\text{setUnion} \ (\text{setWrap} (\text{setUnion}\ K \rightarrow \text{Set}), \text{setWrap} (\text{setUnion}\ K \rightarrow \text{Set}))
\]

RULE
\[
\text{setUnion} \ (\text{setWrap} (\text{setUnion}\ K \rightarrow \text{Set}), \text{setWrap} (\text{setUnion}\ K \rightarrow \text{Set}))
\]

\[
\text{setWrap} (\text{S1})
\]

212
SYNTAX  \( Set ::= \text{getSet} \ (K) \ [\text{function}] \)

RULE
\[
\text{getSet} \ (\text{setWrap} \ (S_1: \text{Set})) \\
\]
\( S_1 \)
the concatenation of two kr[\text{KList}] items

SYNTAX  \( KItem ::= \text{klistConcat} \ (KRLListWrap, KRLListWrap) \ [\text{strict}] \)

RULE
\[
\text{klistConcat} \ (kr[KL_1: KList], kr[KL_2: KList]) \\
kr[KL_1, KL_2] \]
Counts the number of elements in the KList list. Evaluates to an Int.

SYNTAX  \( KItem ::= \text{length} \ (KListWrap) \ [\text{function}] \)
| \( \text{length} \ (\text{Int}, KListWrap) \ [\text{function}] \)

SYNTAX  \( KItem ::= \text{getLastKListElement} \ (KListWrap) \ [\text{function}] \)

RULE
\[
\text{getLastKListElement} \ ([\_ , K:K]) \\
K \]

RULE
\[
\text{length} \ ([Ks:KList]) \\
\text{length} \ (0, [Ks]) \]

RULE
\[
\text{length} \ (I: \text{Int}, [K:K, Ks:KList]) \\
\text{length} \ (I +_{\text{Int}} 1, [Ks]) \]

RULE
\[
\text{length} \ (I: \text{Int}, [\_KList]) \\
I \]

213
Other auxiliary constructs

Generic guard. A generic computational guard (should be builtin): it allows the computation to continue only if a prefix guard evaluates to true.

**SYNTAX** \[ KItem ::= true? \]

**RULE**

\[
\begin{align*}
\text{true :: bool } \land \text{ true?} \\
\end{align*}
\]

\[
\kappa \\
\text{[structural]}
\]

### B.2.3 ClassType functions

Converts a fully qualified class type into a simple name (Id)

**SYNTAX** \[ KItem ::= \text{getSimpleName (ClassType)} \] [function]

**RULE**

\[
\begin{align*}
\text{getSimpleName (class ClassId.Id)} \\
\text{String2Id (trimHead (Id2String (ClassId), rfindString (Id2String (ClassId), ",", lengthString (Id2String (ClassId))) + Int 1))}
\end{align*}
\]

### B.3 Module CORE-CLASSES

This module is a collection of auxiliary functions related to classes and packages.

#### B.3.1 Shortcuts for frequently used classes

Shortcuts for the most frequently used classes that need special treatment within the semantics.

**SYNTAX** \[ KItem ::= \text{class Object} \] [function]

**RULE**

\[
\begin{align*}
\text{class Object} \\
\text{class String2Id ("java.lang.Object")}
\end{align*}
\]

**SYNTAX** \[ KItem ::= \text{class String} \] [function]

**RULE**

\[
\begin{align*}
\text{class String} \\
\text{class String2Id ("java.lang.String")}
\end{align*}
\]
SYNTAX  \( KItem ::= \text{class NullPointerException} \) [function]

RULE

\[
\begin{align*}
\text{class NullPointerException} \\
\text{class String2Id ("java.lang.NullPointerException"} \\
\end{align*}
\]

SYNTAX  \( KItem ::= \text{classArrayImpl} \) [function]

RULE

\[
\begin{align*}
\text{classArrayImpl} \\
\text{class String2Id ("java.lang.ArrayImpl")} \\
\end{align*}
\]

B.3.2 Auxiliary functions for packages

A Java package.

SYNTAX  \( PackageId ::= \text{packageId (Id)} \)

SYNTAX  \( KResult ::= PackageId \)

Converts a term of type ClassType into a term of type PackageId representing this class. This is the package for this class’ inner classes.

SYNTAX  \( KItem ::= \text{toPackage (ClassType)} \) [function]

RULE

\[
\begin{align*}
\text{toPackage ( class Class:Id)} \\
\text{packageId (Class)} \\
\end{align*}
\]

Returns the package of the given class

SYNTAX  \( KItem ::= \text{getPackage (K)} \) [strict]

RULE

\[
\begin{align*}
\text{getPackage ( class Class:Id)} \\
\text{packageId ( String2Id ( retainHead ( Id2String (ClassId), rffindString ( Id2String ( ClassId), ".", lengthString ( Id2String (ClassId))))))} \\
\end{align*}
\]
B.3.3 Auxiliary functions for classes

Converts a pair of PackagId, Id into a fully qualified class name

**SYNTAX**  \( \text{ClassType} ::= \text{getClassType} \ (\text{PackagId, Id}) \)  [function]

**RULE**

\[
\text{getClassType} \ (\text{packageId} (\text{PackKId:Id}), \text{SimpleClass:Id})
\]

\[
\text{class} \ \text{String2Id} \ (\text{Id2String} (\text{PackKId}) + \text{String} "." + \text{Id2String} (\text{SimpleClass}))
\]

Returns the top-level class enclosing this class

**SYNTAX**  \( KItem ::= \text{getTopLevel} \ (\text{ClassType}) \)

**RULE getTopLevel-move-up**

\[
\begin{align*}
& \text{getTopLevel} \ (\text{Class:ClassType}) \\
& \text{getTopLevel} \ (\text{EnclosingClass}) \\
& \langle \text{EnclosingClass:ClassType} \rangle_{\text{enclosingClass}}
\end{align*}
\]

\[
\text{REQUIRES} \ \text{EnclosingClass} \neq \ K \cdot K
\]

**RULE getTopLevel**

\[
\begin{align*}
& \text{getTopLevel} \ (\text{Class:ClassType}) \\
& \langle \text{Class} \rangle_{\text{classType}}
\end{align*}
\]

\[
\text{Class} \langle \langle \ \rangle \rangle_{\langle \ \rangle}_{\langle \ \rangle} \langle \langle \ \rangle \rangle_{\langle \ \rangle}_{\langle \ \rangle}
\]

\[
\text{crntClass}
\]

\[
\text{enclosingClass}
\]

Restore the content of \(<\text{crntClass}>\) with the given class

**SYNTAX**  \( KItem ::= \text{restoreCrntClass} \ (\text{ClassType}) \)

**RULE restoreAfterProcessLocalClass**

\[
\begin{align*}
& \text{restoreCrntClass} \ (\text{Class:ClassType}) \\
& \langle \langle \ \rangle \rangle_{\langle \ \rangle}_{\langle \ \rangle} \langle \langle \ \rangle \rangle_{\langle \ \rangle}_{\langle \ \rangle} \langle \langle \ \rangle \rangle_{\langle \ \rangle}_{\langle \ \rangle}
\end{align*}
\]

Get the internal constructor name for a given class

**SYNTAX**  \( KItem ::= \text{getConsName} \ (\text{ClassType}) \)  [function]

**RULE**

\[
\text{getConsName} \ (\text{class ClassId:Id})
\]

\[
\text{String2Id} ("$cons$" + \text{Id2String} (\text{ClassId}))
\]
B.4 Module PRIMITIVE-TYPES

B.4.1 Integer value normalization

SYNTAX   \( \text{Int} ::= \text{bitCount} \ (\ Type \ ) \ [\text{function}] \)

RULE
\[
\text{bitCount} \ (\ \text{byte} \ ) \\
\hline
8
\]

RULE
\[
\text{bitCount} \ (\ \text{short} \ ) \\
\hline
16
\]

RULE
\[
\text{bitCount} \ (\ \text{int} \ ) \\
\hline
32
\]

RULE
\[
\text{bitCount} \ (\ \text{long} \ ) \\
\hline
64
\]

RULE
\[
\text{bitCount} \ (\ \text{char} \ ) \\
\hline
16
\]

SYNTAX   \( \text{KItem} ::= \text{normalize} \ (\ \text{TypedVal} \ ) \)

RULE
\[
\text{normalize} \ (\ \text{I} : \text{Int} :: \ \text{IntT} : \text{IntType} \ ) \\
\hline
\text{ifAux} \ (\ \text{isInRange} \ (\ \text{I} :: \ \text{IntT} \ ) , \ \text{I} :: \ \text{IntT} , \ \text{normalizeImpl} \ (\ \text{I} :: \ \text{IntT} \ ) \ )
\]
REQUIRES \( \text{IntT} \neq \text{char} \)

RULE
\[
\text{normalize} \ (\ \text{I} : \text{Int} :: \ \text{char} \ ) \\
\hline
\text{ifAux} \ (\ \text{isInRange} \ (\ \text{I} :: \ \text{char} \ ) , \ \text{I} :: \ \text{char} , \ \text{toUnsigned} \ (\ \text{normalizeImpl} \ (\ \text{I} :: \ \text{char} \ )) \ )
RULE

\[
\text{normalize}(\text{I}:\text{Int} :: \text{FloatT}:\text{FloatType}) \quad \frac{}{\text{Int2Float(I)} :: \text{FloatT}}
\]

RULE

\[
\text{normalize}(\text{F}:\text{Float} :: \text{IntT}:\text{IntType})
\]

\[
\text{normalize}(\text{Float2Int}(\text{F}) :: \text{IntT}:\text{IntType})
\]

RULE

\[
\text{normalize}(\text{F}:\text{Float} :: \text{FloatT}:\text{FloatType}) \quad \frac{}{\text{F} :: \text{FloatT}}
\]

Symbolic execution limitation: this construct cannot be [function]

**SYNTAX**

\[
K\text{Item} ::= \text{isInRange} (\text{TypedVal})
\]

RULE

\[
\text{isInRange}(\text{I}:\text{Int} :: \text{byte}) \quad \frac{}{((\text{I} \geq_{\text{Int}} -128) \land_{\text{Bool}} (\text{I} \leq_{\text{Int}} 127))}
\]

RULE

\[
\text{isInRange}(\text{I}:\text{Int} :: \text{short}) \quad \frac{}{((\text{I} \geq_{\text{Int}} -32768) \land_{\text{Bool}} (\text{I} \leq_{\text{Int}} 32767))}
\]

RULE

\[
\text{isInRange}(\text{I}:\text{Int} :: \text{int}) \quad \frac{}{((\text{I} \geq_{\text{Int}} -2147483648) \land_{\text{Bool}} (\text{I} \leq_{\text{Int}} 2147483647))}
\]

RULE

\[
\text{isInRange}(\text{I}:\text{Int} :: \text{long}) \quad \frac{}{((\text{I} \geq_{\text{Int}} -9223372036854775808) \land_{\text{Bool}} (\text{I} \leq_{\text{Int}} 9223372036854775807))}
\]

RULE

\[
\text{isInRange}(\text{I}:\text{Int} :: \text{char}) \quad \frac{}{((\text{I} \geq_{\text{Int}} 0) \land_{\text{Bool}} (\text{I} \leq_{\text{Int}} 65535))}
\]
RULE

isInRange (RV:RawVal :: ---)

true

REQUIRES ¬ Bool (isInt(RV) = R true)

Symbolic execution limitation: this construct cannot be [function]

SYNTAX  KItem ::= normalizeImpl (TypedVal)

RULE

normalizeImpl (I:Int :: T:Type)

normalizeSign (((I &_Int ((1 << _Int bitCount (T)) -_Int 1)) +_Int (1 << _Int bitCount (T)))
&_Int ((1 << _Int bitCount (T)) -_Int 1)) :: T)

SYNTAX  KItem ::= normalizeSign (TypedVal) [strict]

RULE

normalizeSign (I:Int :: T:Type)

ifAux (I ≤_Int ((1 << _Int (bitCount (T) -_Int 1)) -_Int 1), I :: T, I -_Int (1 << _Int bitCount (T)) :: T)

SYNTAX  KItem ::= toUnsigned (TypedVal) [strict]

RULE

toUnsigned (I:Int :: T:Type)

ifAux (I ≥_Int 0, I :: T, I +_Int (1 << _Int bitCount (T)) :: T)

B.4.2 Type normalization

SYNTAX  KItem ::= normalizeType (Type) [function]

RULE

normalizeType (ILT:IntOrLongType)

ILT

RULE

normalizeType (FT:FloatType)

FT
RULE

\[
\text{normalizeType (IT:IntType)}
\]
\[
\text{int}
\]
\text{REQUIRES \( (\text{IT} \neq \text{K\ int}) \land_{\text{Bool}} (\text{IT} \neq \text{K\ long}) \)}

Important! Binary normalizeType cannot be function because it uses subtype in implementation.

SYNTAX \( KItem ::= \text{normalizeType (Type, Type)} \)

RULE

\[
\text{normalizeType (NT1:IntType, NT2:IntType)}
\]
\[
\text{int}
\]
\text{REQUIRES \( (\text{NT1} \neq \text{K\ long}) \land_{\text{Bool}} (\text{NT2} \neq \text{K\ long}) \)}

RULE

\[
\text{normalizeType (NT1:IntType, NT2:IntType)}
\]
\[
\text{long}
\]
\text{REQUIRES \( (\text{NT1} = \text{K\ long}) \lor_{\text{Bool}} (\text{NT2} = \text{K\ long}) \)}

RULE

\[
\text{normalizeType (T:Type, FT:FloatType)}
\]
\text{ifAux (subtype(T, FT), FT, T)}

RULE

\[
\text{normalizeType (FT:FloatType, T:Type)}
\]
\text{ifAux (subtype(T, FT), FT, T)}

RULE

\[
\text{normalizeType (Class:ClassType, ---)}
\]
\text{Class}
\text{REQUIRES \( (\text{Class} = \text{K\ class\ String}) \)}

RULE

\[
\text{normalizeType (---, Class:ClassType)}
\]
\text{Class}
\text{REQUIRES \( (\text{Class} = \text{K\ class\ String}) \)}
RULE

\[\text{normalizeType}(T_1:Type, T_2:Type) \quad \frac{}{T_1} \]

REQUIRES \(\neg_{\text{Bool}} (\text{isNumericType}(T_1) =_{K} \text{true}) \land_{\text{Bool}} (T_2 \neq_{K} \text{class String})\)

RULE

\[\text{normalizeType}(\text{bool, bool}) \quad \frac{}{\text{bool}} \]

B.5 Module SUBTYPING

Checks whether first type is a subtype of the second

SYNTAX \(\text{KItem ::= subtype}(\text{Type}, \text{Type})\) [strict]

RULE subtype-same-type-True

\[\text{subtype}(T:Type, T) \quad \frac{}{\text{true}}\]

[structural]

B.5.1 Subtyping among primitive types

RULE

\[\text{subtype}(\text{byte, T}:Type) \quad \frac{}{(T =_{K} \text{short}) \lor_{\text{Bool}} (T =_{K} \text{int}) \lor_{\text{Bool}} (T =_{K} \text{long}) \lor_{\text{Bool}} (\text{isFloatType}(T) =_{K} \text{true})} \]

REQUIRES \(T \neq_{K} \text{byte}\)

[structural]

RULE

\[\text{subtype}(\text{short, T}:Type) \quad \frac{}{(T =_{K} \text{int}) \lor_{\text{Bool}} (T =_{K} \text{long}) \lor_{\text{Bool}} (\text{isFloatType}(T) =_{K} \text{true})} \]

REQUIRES \(T \neq_{K} \text{short}\)

[structural]

RULE

\[\text{subtype}(\text{int, T}:Type) \quad \frac{}{(T =_{K} \text{long}) \lor_{\text{Bool}} \text{isFloatType}(T) =_{K} \text{true})} \]

REQUIRES \(T \neq_{K} \text{int}\)

[structural]
RULE

\[
\text{subtype} \ (\text{long}, T:Type) \\
\]
\[
is\text{FloatType}(T) =_K \text{true} \\
\]
REQUIRES \(T \neq_K \text{long}\)  
[structural]

RULE

\[
\text{subtype} \ (\text{char}, T:Type) \\
\]
\[
(T =_K \text{int}) \lor \text{Bool}(T =_K \text{long}) \lor \text{Bool}(\text{isFloatType}(T) =_K \text{true}) \\
\]
REQUIRES \(T \neq_K \text{char}\)  
[structural]

RULE

\[
\text{subtype} \ (\text{float}, T:Type) \\
\]
\[
(T =_K \text{double}) \\
\]
REQUIRES \(T \neq_K \text{float}\)  
[structural]

RULE

\[
\text{subtype} \ (\text{double}, T:Type) \\
\]
\[
\text{false} \\
\]
REQUIRES \(T \neq_K \text{double}\)  
[structural]

RULE

\[
\text{subtype} \ (\text{bool}, T:Type) \\
\]
\[
\text{false} \\
\]
REQUIRES \(T \neq_K \text{bool}\)  
[structural]

B.5.2 Subtyping among reference types

The subclass relation introduces a subtyping relation.
RULE SUBTYPE-CLASSOFCLASSRED

\[ \text{subtype} ( \text{Class1}:\text{ClassType}, \text{Class}:\text{ClassType} ) \]
\[ \text{subtype} ( \text{Class2}:\text{ClassType}, \text{Class} ) \]
\[ \langle \langle \text{Class1} \rangle \text{classType} \langle \langle \text{Class2} \rangle \text{extends} \langle \langle \text{class} \rangle \text{classMetaType} \rangle \text{class} \langle \langle \text{Class} \rangle \text{classType} \langle \langle \text{class} \rangle \text{classMetaType} \rangle \text{class} \rangle \]

REQUIRES \text{Class1} \neq_K \text{Class}
[structural]

RULE SUBTYPE-NOCLASSOFANYFALSE

\[ \text{subtype} ( \text{'K}, \text{---} ) \]
\[ \text{false} \]

RULE SUBTYPE-ANYOFNOCLASSFALSE

\[ \text{subtype} ( \text{---}, \text{'K} ) \]
\[ \text{true} \]

RULE SUBTYPE-IINTERFACEOFCLASS

\[ \text{subtype} ( \text{Class1}:\text{ClassType}, \text{Class}:\text{ClassType} ) \]
\[ \text{Class} =_K \text{class Object} \]
\[ \langle \langle \text{Class1} \rangle \text{classType} \langle \langle \text{interface} \rangle \text{classMetaType} \rangle \text{class} \langle \langle \text{Class} \rangle \text{classType} \langle \langle \text{interface} \rangle \text{classMetaType} \rangle \text{class} \rangle \]

RULE SUBTYPE-OFINTERFACE

\[ \text{subtype} ( \text{Class1}:\text{ClassType}, \text{Class2}:\text{ClassType} ) \]
\[ \text{Class2 in ISet} \]
\[ \langle \langle \text{Class1} \rangle \text{classType} \langle \langle \text{ISet}:\text{Set} \rangle \text{implTrans} \rangle \text{class} \langle \langle \text{Class2} \rangle \text{classType} \langle \langle \text{interface} \rangle \text{classMetaType} \rangle \text{class} \rangle \]

RULE SUBTYPE-CLASSOFOTHERFALSE

\[ \text{subtype} ( \text{---}:\text{ClassType}, \text{T}:\text{Type} ) \]
\[ \text{false} \]
REQUIRES \neg_{\text{Bool}} ( is\text{ClassType}(\text{T}) =_K \text{true} )

RULE SUBTYPE-ARRAYOFCLASS

\[ \text{subtype} ( \text{arrayOf} \text{---}, \text{Class}:\text{ClassType} ) \]
\[ \text{Class} =_K \text{class Object} \]
rule subtype-ArrayOfOtherFalse

 subtype ( array0f _, T.Type )

 \[false\]

 requires \((\) getKLabel ( T ) \neq \) KLabel 'arrayOf_' \(\wedge_{\text{Bool}} \neg_{\text{Bool}} (\) isClassType (T) = K true \)

rule subtype-Null

 subtype ( nullType, T.Type )

 \[\text{( isRefType(T) } =_{K} \text{ true )} \]
 [structural]

 Subtype

rule subtype-ArrayOfArrayPrimitive

 subtype ( array0f T1:Type, array0f T2:Type )

 \[\text{( T1 } =_{K} \text{ T2 )} \]

 requires \(\neg_{\text{Bool}} (\) isRefType(T1) =_{K} \text{ true }) \lor_{\text{Bool}} \neg_{\text{Bool}} (\) isRefType(T2) =_{K} \text{ true )} \]

rule subtype-ArrayOfArrayRef

 subtype ( array0f RefT1:RefType, array0f RefT2:RefType )

 subtype (RefT1, RefT2 )

rule subtype-OfNoClass

 subtype ( _, 'K )

 \[false\]

B.5.3 Subtyping lists of types

Checks whether the each type in the first list of types is a subtype of the type at the same position in the second list. If lists have different size, the function will evaluate to false. Used in method call overloading resolution.

syntax \(KItem ::=\) subtypeList ( Types, Types )

rule subtypeList

 subtypeList ( types ( T1:Type, Ts:KList ), types ( Tp1:Type, Tps:KList ))

 andAux ( subtype ( T1, Tp1 ), subtypeList ( types (Ts), types (Tps)) )
 [structural]
RULE subtypeList-EmptyLeft

subtypeList ( types (KList), types (—:Type, —) )

false

[structural]

RULE subtypeList-EmptyRight

subtypeList ( types (—:Type, —), types (KList) )

false

[structural]

RULE subtypeList-EmptyBoth

subtypeList ( types (KList), types (KList) )

true

[structural]

B.6 Module AUX-STRINGS

Utility functions for string manipulation. No dependencies on other K-Java modules. Extension to string.k.

Retain the first Count chars in the string

SYNTAX  KItem ::= retainHead ( String, Int ) [function]

RULE

retainHead ( Str:String, Count:Int )

substrString ( Str, 0, Count )

Retain the last Count chars in the string

SYNTAX  KItem ::= retainTail ( String, Int ) [function]

RULE

retainTail ( Str:String, Count:Int )

substrString ( Str, lengthString ( Str ) — Int Count, lengthString ( Str ) )

Trim the first Count chars in the string

SYNTAX  KItem ::= trimHead ( String, Int ) [function]

RULE

trimHead ( Str:String, Count:Int )

substrString ( Str, Count, lengthString ( Str ) )
Trim the last Count chars in the string

**SYNTAX**  \[ KItem ::= \text{trimTail} \ (String, \ Int) \ [function] \]

**RULE**

\[
\text{trimTail} \ (Str : String, \ Count : Int) \\
\quad \quad \rightarrow \quad \text{substrString} \ (Str, \ 0, \ \text{lengthString} \ (Str) - Int \ Count)
\]

**SYNTAX**  \[ KItem ::= \text{lastChar} \ (String) \ [function] \]

**RULE**

\[
\text{lastChar} \ (Str : String) \\
\quad \quad \rightarrow \quad \text{retainTail} \ (Str, \ 1)
\]
Appendix C

K-Java Static semantics

C.1 Module CONFIGURATION-PREP

The Static semantics consists of several phases that analyze the input program, distribute it from the AST form to a set of cells and finally assembles it back to an AST, this time containing just a subset of features of Java. Each phase digs deeper into a program structure, and most phases also store their result into new cells. Below is the list of global phases, in their execution order:

- Process Type Names
- Process Compilation Units
- Process Class Declarations
- Process Class Members
- Elaboration
- Folding

The configuration cells may be divided into 2 categories: cells directly placed inside the all-enclosing cell $T$ (top-level cells), and cells inside $\text{classes}$. The cell $\text{classes}$ is a collection of $\text{class}$ cells, each representing a Java class (both supported classes from JDK and classes defined by the developer). A $\text{class}$ contains all components of a class — such as extends/implements clauses, imports, fields, methods, etc, each distributed into a separate class, in order to be conveniently accessed when needed.

In the remaining of this section are documented all the cells of the configuration, ordered by the global phase in which they are first used.
Initial state The first row contains three cells most relevant to computation initial state:

- \( \langle K \rangle_k \) — Holds the current computation in all phases of the semantics. Initializes with the AST representation of the program.
- \(\langle K\rangle_{\text{program}}\) — A backup of program AST. Required because the initial AST is needed in both 1st and 2nd phase of static K-Java, but the first phase destroys the content inside \(\langle k\rangle\). Also used in the last phase (Folding) to assemble the preprocessed program.

- \(\langle \text{GlobalPhase}\rangle_{\text{globalPhase}}\) — The current computation global phase.

**Process Type Names** During this phase, one global cell is computed:

- \(\langle \text{Map}[\text{PackageId} \mapsto \text{Map}[\text{Id} \mapsto \text{ClassType}]]\rangle\)\(\text{namesToClasses}\) — A two-level map. First level is a map from package names to second-level maps. Second-level maps are from simple class names to fully qualified class names within the package. This cell is extensively used through the semantics. The map contains both top-level and inner classes. For inner classes, their enclosing package is the fully qualified class name of the directly enclosing class.

Also during Process Type Names classes are first registered. Again, both top-level and inner classes are covered. In a newly created \(\langle \text{class}\rangle\) just a few sub-cells are initialized with data:

- \(\langle \text{ClassType}\rangle_{\text{classType}}\) — The fully qualified class name. The identifier of the class.

- \(\langle \text{ClassMetaType}\rangle_{\text{classMetaType}}\) — Represents whether the type stored in this cell is class or interface. To avoid terminology superfluousness, we will refer hereafter to both classes and interfaces as "classes", making distinctions only when necessary.

- \(\langle \text{ClassAccessMode}\rangle_{\text{classAccessMode}}\) — The access modifier of the class, either public or package.

- \(\langle \text{ClassPhase}\rangle_{\text{classPhase}}\) — Represents the state of this class. In addition to the global computation phase, each class has its own lifecycle phase. Class phases are required to keep track which classes were processed in the current global phase and which were not. During each global phase all classes should transition to a certain class phase. However, as we shall see, not all global phases change the state of the class. The class phases are:
  - Discovered — The initial phase. At the end of Process Type Names all classes are in the state 'Discovered'.
  - Stored
  - Bases Resolved
  - Declaration Processed
  - Members Processed
  - Folded

**Process Compilation Units** At the beginning of this phase computation is again initialized with the initial AST from \(\langle \text{program}\rangle\). The following new cells are filled in inside each \(\langle \text{class}\rangle\):

- \(\langle \text{ClassType}\rangle_{\text{enclosingClass}}\) — The directly enclosing class, for inner classes, or no value for top-level classes.
- \( \{ K \}_{\text{rawExtends}} \) — The extends clause of this class, in its raw (AST) form.

- \( \{ K \}_{\text{rawImplements}} \) — The implements clause, in AST form.

- \( \{ K \}_{\text{rawDeclarations}} \) — The class body, in AST form.

- \( \{ \text{Map}[\text{Id} \rightarrow \text{ClassType}] \}_{\text{cuImports}} \) — A map from names accessible inside this class to fully qualified class names they represent. Only computed for top-level classes at this phase. For inner classes this cell remains empty.

- \( \{ \text{ContextType} \}_{\text{classContextType}} \) — Either static or instance, for inner classes. Always static for top-level classes.

The class phase changes from Discovered to Stored. As we can see, the cells computed so far contain all the data of the original program. Thus, initial AST representation of the program is no longer needed. In fact, the cell \( \{ \text{program} \} \) is discarded at the end of Process Compilation Units. The remaining preprocessing phases will use class data in this initial form to compute other cells within \( \{ \text{class} \} \), finally used for execution.

Also during Process Compilation Units the following global cell is used:

- \( \{ \text{Map}[\text{Id} \rightarrow \text{ClassType}] \}_{\text{compUnitImports}} \) — A map from all type names accessible in the current compilation unit (Java file) to their respective fully qualified class names. This includes both classes accessible through imports declarations and classes declared in the package of the current compilation unit. Used to compute \( \{ \text{cuImports} \} \) of top-level classes.

**Process Class Declarations** Here each class passes through two more class phases: Bases Processed and Declarations Processed. First, for each class the semantics attempts to resolve its extends/implements clauses into fully qualified class names. The order in which dependencies are resolved depends on both class inheritance relationships as well as nesting relationships. Once the dependencies of a class are resolved, they are stored into a temporary cell:

- \( \{ K \}_{\text{unprocessedBases}} \) — Initialized with the list of fully qualified class names for classes mentioned in extends/implements clauses of this class.

Once the content of \( \{ \text{unprocessedBases} \} \) is created, the class enters into Bases Resolved phase. It then waits in this phase until all classes referred in extends/implements reach the phase Declarations Processed. The restrictions in JLS related to class dependencies guarantee that classes cannot have cyclic dependencies, thus a class cannot get locked in the waiting state. The cell \( \{ \text{unprocessedBases} \} \) is used to determine the moment when the class may exit the waiting state. Once a class reaches the phase Declarations Processed, is is deleted from cells \( \{ \text{unprocessedBases} \} \) of other classes. Thus, when all extends/implements dependencies of a class reach the phase Declarations Processed, the content of the its \( \{ \text{unprocessedBases} \} \) cell becomes empty. Once in this state, the class enters into the phase Declarations Processed itself and computes three more cells:

- \( \{ \text{ClassType} \}_{\text{extends}} \) — The base class, fully qualified.

- \( \{ \text{Set}[\text{ClassType}] \}_{\text{implements}} \) — The list of directly implemented interfaces, fully qualified.
\(\{\text{Map[Id} \rightarrow \text{ClassType}\}\}\) \textbf{imports} — The map of classes accessible by simple name within the body of this class. The rules for computing this map are complex and include the following sources:

- Imports declarations of the current compilation unit.
- Classes declared within the package of the current compilation unit.
- Classes accessible within the body of the directly enclosing class, if the current class is inner class.
- Inner classes inherited from base classes, e.g. from extends/implements clauses.
- Inner classes of this class itself.

The need to cover all these cases leads to the intricate order in which class dependencies have to be be resolved.

When a class enters the phase Declarations Processed, the cells \(\{\text{rawExtends}\}, \{\text{rawImplements}\}\) and \(\{\text{unprocessedBases}\}\) are no longer needed and are discarded. Once all classes reach this phase the computation proceeds to the next global phase.

During Process Class Declarations the following global cell is first used:

\(\{\text{ClassType}\}\) \textbf{crntClass} — The current class. Used in multiple phases starting from Process Class Declarations.

\textbf{Process Class Members} During this phase each class processes its members and reaches the state Members Processes. Until then, the class body is stored in \(\{\text{rawDeclarations}\}\). A class member could be one of:

- field
- method
- constructor
- static or instance initializer

The following new class cells are produced:

\(\{\text{Set[ClassType]}\}\) \textbf{implTrans} — The transitive closure of implemented interfaces. In the remaining phases this set is used by the subtyping relationship.

\(\{\text{Map [ Signature} \rightarrow \text{ClassType}\}\}\) \textbf{methods} — The map of accessible methods. Keys are method signatures, values are classes where methods are defined. Includes both methods declared within this class as well as methods inherited from base classes/ base interfaces.

\(\{\text{Bag}\}\) \textbf{methodDecs} — The collection of method declarations (\(\{\text{methodDec}\}\) cells) in the current class. This cell contains only a subset of methods from \(\{\text{methods}\}\), as the set of accessible methods from \(\{\text{methods}\}\) also includes methods inherited from base classes/interfaces. Hence the need of two separate collections. Each \(\{\text{Bag}\}\) \textbf{methodDec} contains the following data:
- \( \langle \text{Signature} \rangle \text{methodSignature} \) — The method signature, acting as identifier of the \( \langle \text{methodDec} \rangle \).
- \( \langle \text{Type} \rangle \text{methodReturnType} \) — The method return type.
- \( \langle \text{List[ Param]} \rangle \text{methodParams} \) — The method parameters.
- \( \langle K \rangle \text{methodConstrFirstLine} \) — the first line of a constructor (if this method is indeed a constructor, for other classes than \text{Object}). It contains a call to another constructor: either \text{super()} or \text{this()}.
- \( \langle K \rangle \text{methodBody} \) — The method body.
- \( \langle \text{AccessMode} \rangle \text{methodAccessMode} \) — The method access mode.
- \( \langle \text{ContextType} \rangle \text{methodContextType} \) — May be either static or instance.
- \( \langle \text{MethodMetaType} \rangle \text{methodMetaType} \) — May be either method or constructor.

- \( \langle K \rangle \text{instanceFields} \) — The list of instance field declarations, stored as a list of local variable declaration statements, without initializers. Used during object instantiation.

- \( \langle K \rangle \text{instanceInit} \) — The list of instance initializers of the class combined into one big instance initializer. Instance field initializers are also concatenated into this cell in their textual order.

- \( \langle K \rangle \text{staticFields} \) — The list of static field declarations, in a similar format to \( \langle \text{instanceFields} \rangle \).

- \( \langle K \rangle \text{staticInit} \) — The list of static initializers and static field initializers concatenated into one block.

- \( \langle \text{Map[Id} \mapsto \text{Value}] \rangle \text{constantEnv} \) — The map from compile-time constants to their actual values. Constants in Java have a slightly different semantics compared to final static fields. In particular, accessing them don’t trigger static initialization of the declaring class.

Once all the cells above are computed the class proceeds into the phase Members Processed and the cell \( \langle \text{rawDeclarations} \rangle \) is deleted.

**Elaboration** Here all the code blocks are processed — method and constructor bodies, static and instance initializers. Most of the information traditionally inferred by the compiler is computed at this phase. More precisely the elaboration performs the following transformations:

- Each name is resolved into local variable, field, method, class or package. While a method may be distinguished from other categories purely syntactically, resolving to other categories requires knowledge of the names existing in the current context.

- Simple class names are resolved into fully qualified class names. Hereafter all the class names in the code are fully qualified.

- The compile-time type of each expression is inferred. Thus, when the code reaches execution phase, expressions are no longer in their initial form. The expressions are annotated with their types.

- For each method call the precise signature in inferred.
Local and anonymous classes are processed. The earliest phase where local classes could be discovered is elaboration. Still, local classes have all the features of other classes. Thus they need to be passed through all the preprocessing steps. The whole preprocessing for local classes is performed during the global phase elaboration.

Despite this phase being the most complex preprocessing phase of all, it introduces few new cells. Most of them are related to local classes. Inside \{\text{class}\} just one new cell is introduced:

- \{\text{Map}[\text{Id} \rightarrow \text{Type}]\}_\text{enclosingLocalEnv} — The map from local variables of the current block to their types. Used during local classes processing.

Among global cells the following new cells are added:

- \{\text{List}[\text{mapWrap(Map}[\text{Id} \rightarrow \text{Type}])]\}_\text{elabEnv} — A stack where each layer is a map of local variables. Each layer of the stack represents a code block, in the blocks nesting order. Inside each layer, the map is from local variables accessible in that layer to variable types.

- \{\text{Int}\}_\text{contextType} — The context type of the currently elaborated class. Either static or instance.

- \{\text{List}[\text{mapWrap(Map}[\text{Id} \rightarrow \text{ClassType}])]\}_\text{localTypes} — A cell similar in structure to \{\text{elabEnv}\}. This time it contains stack layers which are maps from local class names to local class types.

- \{\text{Int}\}_\text{nextLocalId} — A number used to generate unique fully-qualified class names for local classes.

- \{\text{K}\}_\text{elabBuffer} — A temporary cell used during elaboration of local classes.

During the elaboration phase no cells are consumed. Instead, the code blocks stored inside \{\text{methodDecs}\}, \{\text{instanceInit}\}, \{\text{staticInit}\} are processed and stored back into the same cell. After elaboration the classes remain in the same state — Members Processed. The state Members Processed is in fact the final state of the cell \{\text{class}\}.

**Folding** During the last phase of static semantics the program is assembled from \{\text{classes}\} back into the AST form. First the content of each class is assembled into \{\text{K}\}_\text{folded} of the respective class, and the class phase changes to Folded. Second, the AST representation of each class is appended into \{\text{program}\}. When this phase ends, the content of \{\text{program}\} is printed to the standard output.

### C.2 Module PROCESS-TYPE- NAMES

First pass - collecting globally accessible names in the program. This includes packages, top-level classes and inner classes. In each compilation unit just class declarations (both global and inner) are processed. Results are stored in the cell <namesToClasses>.

The initial configuration contains the initial program in cells <k> and <program>, and global phase is ProcTypeNamesPhase.
C.2.1 Compilation units

**rule CompilationUnit-DefaultPackage-Desugar**

\[ '\text{CompilationUnit}(\quad\quad\quad\quad\quad\quad) \]

\[ '\text{None}(\quad) \]

\[ '\text{Some}(\text{PackageDec}(\_K, \text{PackageName}[\_K\text{List}])) \]

**CONTEXT**

\[ '\text{CompilationUnit}(\text{Some}(\text{PackageDec}(\quad:K, \Box)), \quad) \]

**rule CompilationUnit**

\[ '\text{CompilationUnit}(\text{Some}(\text{PackageDec}(\quad:K, \text{Pack:PackageId})), \quad:K, [\text{Decs:KList}]) \]

\[ \text{processTypeNames}([\text{Decs}], \text{Pack}) \]

\[ \langle \text{ProcTypeNamesPhase} \rangle_{\text{globalPhase}} \]

C.2.2 Package declarations

**rule PackageName-Start**

\[ '\text{PackageName}([\text{Ks:KList}]) \]

\[ \text{packageNameImpl}(\quad) \]

\[ [\text{structural}] \]

**SYNTAX**

\[ K\text{Item} ::= \text{packageNameImpl}(\quad\text{String}, \text{KListWrap}) \]

**rule packageNameImpl-FirstInnerDesugar**

\[ \text{packageNameImpl}(\quad:\text{String}, [\quad\text{Id}:\text{Id}, \text{Ks:KList}]) \]

\[ [\text{structural}] \]

**rule packageNameImpl-FirstPack**

\[ \text{packageNameImpl}(\quad, [\quad\text{packageId}(\quad\text{Id}):\text{Id}, \text{Ks:KList}]) \]

\[ \text{packageNameImpl}(\quad\text{Id}2\text{String}(\quad\text{NextKId}:\text{Id}), [\quad\text{Ks}]) \]

\[ [\text{structural}] \]

**rule packageNameImpl-NextPack**

\[ \text{packageNameImpl}(\quad\text{Str}+\text{String} \_\quad+\text{String}\text{Id}2\text{String}(\quad\text{NextKId}), [\quad\text{Ks}]) \]

\[ \text{requires} \text{Str}!\neq\text{String} \_\quad \]

\[ [\text{structural}] \]
RULE `packageNameImpl-End`

```
packageNameImpl (Str: String, [\`KList \])
```

```
packageNameImpl (String2Id (Str))
```

[structural]

C.2.3 Class declaration

Wrapper of declarations inside a compilation unit. Used to distinguish ProcTypeNamesPhase from ProcCom-
pUnitsPhase

SYNTAX  

```
KItem ::= processTypeNames (KListWrap, PackageId)
```

RULE `processTypeNames-AddPackage`

```
\{ processTypeNames (\ldots, PackId:PackageId) \ldots \}_k
```

```
\{ PackMap:Map

\{ PackMap [ mapWrap (\`Map\) / PackId \]
```

REQUIRES  

```
\neg Bool PackId in keys (PackMap)
```

A type name is either class or interface name. Anonymous labels will be one of:

- \`ClassDec\`(`ClassDecHead(\ldots))
- \`InterfaceDec\`(`InterfaceDecHead(\ldots))

RULE `processTypeNames`

```
processTypeNames ( \[ 
DecLabel: KLabel(\ldots:KLabel (Modifiers:KListWrap, SimpleClass:Id, \ldots),

CBODY:K)
\], \ldots, PackId:PackageId) ↦
```

```
\{ processInnerTypes (CBODY, toPackage (getClassType (PackId,

SimpleClass)))
```

```
\ldots \rightarrow mapWrap ( 
```
\{ ClassesMap:Map

\{ ClassesMap [ getClassType (PackId, SimpleClass) /

SimpleClass ]
```

```
\ldots \rightarrow Bag
```

```
\{ getClassType (PackId, SimpleClass) \} [ classType

\{ getMetaType (DecLabel) \} [ classMetaType (DiscoveredCPhase) [ classPhase \ldots ] ]
```

REQUIRES  

```
DecLabel = KLabel \`ClassDec \lor Bool DecLabel = KLabel \`InterfaceDec
```

235
**RULE processTypeNames-ElemDiscard**

\[
\text{processTypeNames} ( \text{Label} : K\text{Label}(\ldots), \ldots, \ldots, \ldots )
\]

**REQUIRES** \( \neg \text{Bool} ( ( \text{Label} = K\text{Label} \ 'ClassDec) \lor \text{Bool} ( \text{Label} = K\text{Label} \ 'InterfaceDec) ) \)

We match PackId in \( \text{namesToClasses} \) just to be sure that the package was added to the map.

**RULE processTypeNames-Discard**

\[
\begin{align*}
\text{processTypeNames} & \quad ( \ldots, K\text{List}, \text{PackId}: \text{PackageId}) \\
& \quad \vdash K \langle \ldots \text{PackId} \mapsto \ldots \rangle \text{namesToClasses}
\end{align*}
\]

Is converted into a processTypeNames. The first argument is a class/interface body. This function have to extract the list of members from the body and pass them to processTypeNames.

**SYNTAX** \( K\text{Item} ::= \text{processInnerTypes} ( K, \text{PackageId} ) \)

**RULE**

\[
\text{processInnerTypes} ( '\text{ClassBody}([ \text{Ks}: \text{KList} ]), \text{Pack}: \text{PackageId} )
\]

\[
\vdash \text{processTypeNames} ( [ \text{Ks} ], \text{Pack} )
\]

**RULE**

\[
\text{processInnerTypes} ( [ \text{Ks}: \text{KList} ], \text{Pack}: \text{PackageId} )
\]

\[
\vdash \text{processTypeNames} ( [ \text{Ks} ], \text{Pack} )
\]

### C.2.4 Auxiliary constructs

Computes the metaType based on type declaration label

**SYNTAX** \( K\text{Item} ::= \text{getMetaType} ( K\text{Label} ) \) [function]

**RULE**

\[
\text{getMetaType} ( '\text{ClassDec} )
\]

\[
\text{class}
\]

**RULE**

\[
\text{getMetaType} ( '\text{InterfaceDec} )
\]

\[
\text{interface}
\]

Returns the types map for the given package, or mapWrap(.Map) if there’s no such package.

**SYNTAX** \( K\text{Item} ::= \text{getNamesMap} ( \text{PackageId} ) \)

236
C.3 Module PROCESS-COMP-UNITS

Module Overview
For each Compilation unit:

- Process import declarations. Build <compUnitImports>.

For each class C in CompUnit:

- Add to <class> containing C the cells mentioned in configuration doc.
- Process inner classes of C.

rule ProcCompUnitsPhase-start

\[
\langle \text{Program:K} \rangle \rightarrow^k \langle \text{ProcCompUnitsPhase} \rangle_{\text{globalPhase}}
\]

Import declarations are processed in the module PROCESS-IMPORTS

rule CompilationUnit

\[
\langle '\text{CompilationUnit('Some('PackageDec(\ldots:K, \text{Pack:PackageId}) \)), \text{ImpDecs:K,} [\text{Ks:KList}] \rangle \rightarrow^k \langle \text{compUnitImportsStart (Pack) \ \text{impDecs} \ \text{processTypeDecsInPCUPhase (} [\text{Ks}], \text{Pack, }'K \}) \rangle_{\text{globalPhase}} \rangle
\]

Wrapper over a list of type declarations and possible other terms, required to distinguish this phase from other preprocessing phases Type declarations are processed in this phase wrapped in a processTypeDecsInPCUPhase

SYNTAX

\[ KItem ::= \text{processTypeDecsInPCUPhase (} KListWrap, \text{PackageId}, \text{ClassType} \) \]
rule processTypeDecsInPCUPhase-Expand

\[ \forall K \]
processTypeDecsInPCUPhase ( \[ K1, Pack, OuterClass \])
\[ \leadsto \] processTypeDecsInPCUPhase ( \[ K1.K, -, -, Pack: PackageId, OuterClass: ClassType \])

rule processTypeDecsInPCUPhase-Discard

processTypeDecsInPCUPhase ( \[ 'List, -, - \])
\[ \forall K \]

rule processTypeDecsInPCUPhase-ElemDiscard

processTypeDecsInPCUPhase ( \[ Label:KLabel( - )], -, - )
\[ \forall K \]
REQUIRES \( \forall_{Bool} ( Label =_{KLabel} 'ClassDec \lor_{Bool} Label =_{KLabel} 'InterfaceDec ) \)

rule processTypeDecsInPCUPhase-typeDec-ComputeFullName

processTypeDecsInPCUPhase ( \[ KL:KLabel( - ),\]
\[ SimpleClass:Id \]
\[ getClassType ( Pack, SimpleClass ) \]
\[ [\text{structural}] \]

Structure of 'ClassDec:

'ClassDec(
'ClassDecHead(
 [Modifiers],,
 Class:ClassType,,
 'None( 'List ),,
 'Some('SuperDec( BaseClassRaw - extends declaration ) ),,
 'Some('ImplementsDec([ ImplTypesRaw - implements declarations ]))
),,
 'ClassBody( [ Decls --- member declarations ] )
)

rule ClassDec-NoExtendsDesugar

processTypeDecsInPCUPhase ( \[ 'ClassDec('ClassDecHead(\[ - : KList Wrap, Class:ClassType,\]
\[ - : K,\]
\[ 'None( - )\]
\[ 'Some('SuperDec(\[ #if ( Class \neq K class Object ) \#then 'ClassType(class Object, 'None(\[ 'List \)) \#else 'K fi))
\[ - : K), \[ - : K \] ), Pack: PackageId, - ) \]
\[ [\text{structural}] \]

238
RULE processTypeDecsInPCUPhase-IIInterfaceDec

\[
\text{processTypeDecsInPCUPhase} \ ( [ '\text{InterfaceDec}'( '\text{InterfaceHead}'( Class::ClassType, ---K, 'Some'( '\text{ExtendsInterfaces}'( ImplTypesRaw::K ) ) ), [ Decls::KList ] ) ), Pack::PackageId, OuterClass::ClassType ) \] \]

\[
\text{saveImportsInPCUPhase} ( \text{Class} ) \leftarrow \text{processTypeDecsInPCUPhase} ( [ \text{Decls} ] , \text{toPackage} ( \text{Class} ), \text{Class} )
\]

If the given type is top-level, then save <imports> from the <compUnitImports> cell. Otherwise do nothing.

SYNTAX \( K\text{Item} ::= \text{saveImportsInPCUPhase} ( K ) \)

RULE saveImportsInPCUPhase

\[
\text{saveImportsInPCUPhase} ( \text{Class::ClassType} ) \] \]

\[
\text{CUImports::Map } \text{compUnitImports}
\]

RULE saveImportsInPCUPhase-Discard

\[
\text{saveImportsInPCUPhase} ( \text{Class::ClassType} ) \] \]

\[
\text{class} \rightarrow \text{enclosingClass}
\]

C.4 Module PROCESS-IMPORTS

Rules related to processing import declarations. Part of ProcCompUnitsPhase.

SYNTAX \( K\text{Item} ::= \text{compUnitImportsStart} ( \text{PackageId} ) \)

RULE compUnitImportsStart

\[
\text{compUnitImportsStart} ( \text{Pack::PackageId} ) \] \]

\[
\text{NameToClasses::Map } \text{compUnitImports}
\]

\[
\leftarrow \text{mapWrap} ( \text{PackMap::Map} ) \] \]

240
CONTEXT

′TypeImportDec(□)

RULE TypeImportDec

′TypeImportDec(Class:ClassType) → k

Imp:Map

Imp [ Class / X ]

compUnitImports

CONTEXT

′TypeImportOnDemandDec(□)

RULE TypeImportOnDemandDec

′TypeImportOnDemandDec(Pack:PackageId) → k

importOnDemandImpl(PackMap)

← Pack ↦ mapWrap(PackMap:Map) → namesToClasses

Imp:Map

Imp [ Class / X ]

compUnitImports

Imports to <compUnitImports> cell public classes from the given map. Classes with package access are ignored.

SYNTAX

KItem ::= importOnDemandImpl(Map)

RULE importOnDemandImpl-public

importOnDemandImpl(X:Id ↦ Class:ClassType) → k

Imp:Map

Imp [ Class / X ]

compUnitImports

Class → classType

public → classAccessMode

RULE importOnDemandImpl-package

importOnDemandImpl(X:Id ↦ Class:ClassType) → k

package → classAccessMode

RULE importOnDemandImpl-discard

importOnDemandImpl(‘Map)

′K

Importing a nonexistent package has no effect. This is required because some tests import some packages from JDK that are not included in class-lib.
C.5  Module PROCESS-CLASS-DECS

C.5.1  Initiate the resolving of class bases

Question: how do we know if bases were already resolved for this class or not? Answer: When resolveBases(Class) is consumed by the rule [resolveBasesEnd], the class changes its state: StoredCPhase => BasesResolvedCPhase
C.5.2 Resolve bases

Process the given class from StoredCPhase to BasesResolvedCPhase. Resolve the base class (arg 2) and interfaces (arg 3) of a class (arg 1).

**Syntax**  \[
KItem ::= \text{resolveBases} \left( \text{ClassType} , K , KListWrap \right)
\]

**Context**

\[
\text{resolveBases} \left( , \Box , \Box \right)
\]

**Rules**

**Rule resolveBases**

\[
\text{resolveBases} \left( \text{Class:ClassType} , \text{BaseClass:ClassType} , \left[ \text{ImplTypes:KList} \right] \right) \xrightarrow{\kappa} \text{processClassDecs-resolveBases-Inner-Start}
\]

\[
\text{resolveBases} \left( \text{Class, RawExtends, } \left[ \text{RawImplements} \right] \right) \xrightarrow{\kappa} \text{resolveBases} \left( \text{ClassType, K} \right)
\]

\[
\text{processClassDecs-resolveBases-Inner-Start} \\
\xrightarrow{\kappa} \text{restoreCrntClass} \left( \text{OldCrntClass} \right)
\]

\[
\xrightarrow{\kappa} \text{processClassDecs} \left( \_ \right)
\]

\[
\text{OldCrntClass:ClassType} \\
\xrightarrow{\kappa} \text{crntClass} \xrightarrow{\_ \text{Map}} \text{compUnitImports}
\]

\[
\text{OuterClass} \\
\langle \text{Class:ClassType} \rangle \langle \text{RawExtends:K} \rangle \langle \left[ \text{RawImplements:KList} \right] \rangle \\
\langle \text{ClassPhase} \rangle \langle \text{RawImplements:KList} \rangle \\
\langle \text{enclosingClass:ClassType} \rangle \\
\langle \text{oldCrntClass:ClassType} \rangle \\
\langle \text{BaseClass:ClassType} \rangle \\
\langle \text{unprocessedBases:KList} \rangle \\
\langle \text{StoredCPhase:ClassPhase} \rangle \\
\langle \text{BasesResolvedCPhase:ClassPhase} \rangle \\
\langle \text{classPhase:ClassPhase} \rangle
\]

**REQUIRES**

\[
\text{OuterClassPhase} =_K \text{DecsProcessedCPhase} \lor \text{Bool} \cdot \text{OuterClassPhase}
\]

\[
=K \text{MembersProcessedCPhase}
\]
C.5.3 Processing after bases were resolved

RULE processClassDecs-mark-Base-Processed

\[ \langle \text{processClassDecs} (\quad) \rangle_k \left( \left[ \text{BaseClass:ClassType} \right] \left[ \text{unprocessedBases} \right] \right) \left( \text{class} \right) \]

\[ \left( \left\{ \text{BaseClass} \right\}_{\text{classType}} \left\{ \text{BaseClassPhase:ClassPhase} \right\}_{\text{classPhase}} \right) \left( \text{class} \right) \]

REQUIRES BaseClassPhase =_K \text{DecsProcessedCPhase} \lor \text{Bool} \text{BaseClassPhase}

This is for class Object

RULE processClassDecs-mark-noClass-Processed

\[ \langle \text{processClassDecs} (\quad) \rangle_k \left( \left[ \text{KList} \right] \left[ \text{unprocessedBases} \right] \right) \left( \text{class} \right) \]

RULE processClassDecs-end

\[ \text{processClassDecs} ( \text{setWrap} ( \text{Class:ClassType} \ \text{RestClasses:Set} ) ) \]

\[ \text{saveImplements} ( \text{Class}, \text{kListToSet} ( \left[ \text{ImplTypes}, \text{Set} \right] ) ) \land \text{saveImports} ( \text{Class}, \text{OuterClass}, \left[ \text{BaseClass}, \text{ImplTypes} \right], \text{K}, \text{mapWrap} ( \text{Map}, \text{K} ) ) \]

\[ \left( \text{Class} \right)_{\text{classType}} \left( \text{OuterClass:ClassType} \right)_{\text{enclosingClass}} \left( \text{CBody:K} \right)_{\text{rawDeclarations}} \]

\[ \left( \text{MetaT:ClassMetaType} \right)_{\text{classMetaType}} \left( \text{BasesResolvedCPhase} \ 	ext{DecsProcessedCPhase} \right)_{\text{classPhase}} \left( \text{BaseClass} \right)_{\text{extends}} \]

\[ \left( \left[ \text{KList} \right] \right)_{\text{unprocessedBases}} \]

RULE processClassDecs-discard

\[ \text{processClassDecs} ( \text{setWrap} ( \text{Set} ) ) \]

\[ \text{'K} \]

Converts a term of type KList into a Set

SYNTAX \text{KItem} ::= \text{kListToSet} ( \text{KListWrap}, \text{Set} )

RULE

\[ \text{kListToSet} ( \left[ \text{KList} \right], \left( \left[ \text{Set} \right] \right) ) \]
RULE

\[
\text{kListToSet} \left( \left[ KList \right], \text{TypeSet:Set} \right) \rightarrow \text{setWrap} \left( \text{TypeSet} \right)
\]

Receives a kListToSet term ans saves the resulting set into <imports>

SYNTAX \[ KItem ::= \text{saveImplements} \left( \text{ClassType}, K \right) \text{ [strict(2)] } \]

RULE \text{SAVEIMPLEMENTS}

\[
\begin{align*}
\text{saveImplements} \left( \text{Class:ClassType}, \text{setWrap} \left( \text{ImplSet:Set} \right) \right) & \rightarrow^{\cdot} K \left\langle \text{Class} \right\rangle_{\text{classListType}} \\
\text{ImplSet} & \rightarrow^{\cdot} \text{implements}
\end{align*}
\]

C.5.4 Computing imports map

Computes the full names map used to resolve classes by simple name inside the class specified by the arg 4, and stores them inside <imports>.

SYNTAX \[ KItem ::= \text{saveImports} \left( \text{ClassType}, \text{ClassType}, \text{KListWrap}, K, K, K \right) \text{ [strict(4,5,6)] } \]

Compute outer classes if this class is a top-level class

RULE \text{SAVEIMPORTS-COMPUTEEXTERNALTOP}

\[
\begin{align*}
\text{saveImports} \left( \text{Class:ClassType}, \cdot, \cdot, \cdot, \text{mapWrap} \left( \text{ExternalMap} \right) \right) & \rightarrow^{\cdot} K \left\langle \text{Class} \right\rangle_{\text{classListType}} \\
\left\langle \text{Class} \right\rangle_{\text{classListType}} & \rightarrow^{\cdot} \text{mapWrap} \left( \text{ExternalMap} \right) \left\langle \text{cuImports} \right\rangle \\
\end{align*}
\]

Compute outer classes if this class is an inner class.

RULE \text{SAVEIMPORTS-COMPUTEEXTERNALINNER}

\[
\begin{align*}
\text{saveImports} \left( \cdot, \text{OuterClass:ClassType}, \cdot, \cdot, \text{mapWrap} \left( \text{ExternalMap} \right) \right) & \rightarrow^{\cdot} K \left\langle \text{OuterClass} \right\rangle_{\text{classListType}} \\
\left\langle \text{OuterClass} \right\rangle_{\text{classListType}} & \rightarrow^{\cdot} \text{mapWrap} \left( \text{ExternalMap} \right) \left\langle \text{imports} \right\rangle
\end{align*}
\]

245
**RULE saveImports-InheritTypes**

\[
\text{saveImports} \left( \cdots, \cdots, \left[ \begin{array}{c}
\text{BaseClass:ClassType} \\
\text{'KList}
\end{array} \right], \cdots, \cdots \right) \\
\text{mapWrap} \left( \text{InheritedMap:Map} \right) \\
\text{mapUnion} \left( \text{mapWrap} \left( \text{InheritedMap} \right), \text{mapWrap} \left( \text{BaseClassMap} \right) \right) \\
\langle \text{BaseClass} \rangle_{\text{classType}} \langle \text{BaseClassMap:Map} \rangle_{\text{imports}}
\]

**RULE saveImports-Discard-noClass**

\[
\text{saveImports} \left( \cdots, \cdots, \left[ \begin{array}{c}
\text{'K} \\
\text{'KList}
\end{array} \right], \cdots, \cdots, \cdots \right)
\]

Compute the final result — the combined accessible classes map, with this components in order:

- external names
- local block names (for local classes only)
- inherited names
- inner names

**RULE saveImports-ComputeResult**

\[
\text{saveImports} \left( \text{Class:ClassType}, \cdots, \left[ \begin{array}{c}
\text{'KList} \\
\text{'K}
\end{array} \right], \cdots \right) \\
\text{mapWrap} \left( \text{ExternalMap:Map} \right), \text{mapWrap} \left( \text{InheritedMap:Map} \right), \\
\text{mapUnion} \left( \text{mapUnion} \left( \text{mapWrap} \left( \text{ExternalMap} \right), \text{mapWrap} \left( \text{LocalTypes} \right) \right), \\
\text{mapUnion} \left( \text{mapWrap} \left( \text{InheritedMap} \right), \text{getNamesMap} \left( \text{toPackage} \left( \text{Class} \right) \right) \right) \right) \\
\langle \text{mapWrap} \left( \text{LocalTypes:Map} \right) \rangle_{\text{localTypes}} \langle \text{BaseClass:ClassType} \rangle_{\text{classType}} \\
\langle \text{BaseClassMap:Map} \rangle_{\text{imports}}
\]

Save the computed Imports map into the <imports> cell of the given class

**RULE saveImports-end**

\[
\text{saveImports} \left( \text{Class:ClassType}, \cdots, \cdots, \cdots, \cdots, \text{mapWrap} \left( \text{Imports:Map} \right) \right) \\
\text{mapWrap} \left( \text{Imports:Map} \right) \\
\langle \text{Class} \rangle_{\text{classType}} \langle \text{Imports} \rangle_{\text{imports}} \langle \text{culImports} \rangle_{\text{'Bag}}
\]

**C.6 Module PROCESS-CLASS-MEMBERS**

We need to process Object first. Thus when we will process any interfaces, Object class will already be processed.
C.6.1 Triggering the processing of depending types

rule processClasses
\( \sim \) processClasses ( setWrap ( ClassType ) )

rule processClasses-Discard
processClasses ( setWrap ( ?Set ) )

rule processTypeWithDepends
\( \sim \) processTypeWithDepends ( Class:ClassType )
\( \sim \) processType ( BaseClass )
\( \sim \) processClasses ( getInnerClasses ( Class ) )
\( \sim \) processTypeWithDepends ( BaseClass:ClassType )
\( \sim \) processClasses ( getTopLevelClasses )
\( \sim \) processClasses ( getTopLevelClasses )

rule processTypeWithDepends-Discard
processTypeWithDepends ( Class:ClassType )

rule processTypeWithDepends-Discard2
processTypeWithDepends ( \( \kappa \) )

SYNTAX  \( KItem ::= \) processClasses ( \( K \) ) [strict]
C.6.2 Initiating class processing

For each class computes the set of inherited interfaces, inherits all the members, processes inner declarations. Computes the following class cells:

- `<implTrans>`
- `<instanceFields>`
- `<methods>`
- `<methodDecs>`
- `<instanceInit>` — temporary cell, deleted after class body is processed.
- `<staticEnv>`
- `<staticInit>`

First we inherit methods from interfaces, then from the base class, and in the end we add methods declared in this class. Each new method overwrites previous methods with the same signature.

**Syntax**

\[ KItem ::= \text{processType} \ (\text{ClassType}) \]

**Rule processType**

\[
\begin{align*}
\text{processType} \ (\text{Class} : \text{ClassType}) \\
\quad \text{computeImplTrans} \ (\text{BaseClass ISet}) \leftrightsquigarrow \text{tryInheritSet} \ (\text{ISet}) \leftrightsquigarrow \text{ifAux} \ ((\text{MetaT} =_K \text{interface}), \text{tryInherit} \ (\text{class Object}), \ 'K) \leftrightsquigarrow \text{tryInherit} \ (\text{BaseClass}) \leftrightsquigarrow \text{Decls} \leftrightsquigarrow \text{convertInstanceInitIntoMethod} \leftrightsquigarrow \text{restoreCrntClass} \ (\text{OldCrntClass}) \\
\end{align*}
\]

**Rule processType-discard**

\[
\begin{align*}
\text{processType} \ (\text{Class} : \text{ClassType}) \\
\quad 'K \quad \text{k} \quad \text{k} \quad \text{k} \\
\end{align*}
\]

C.6.3 Inheriting base types

Compute `<implTrans>` cell — interfaces transitively implemented

**Syntax**

\[ KItem ::= \text{computeImplTrans} \ (\text{Set}) \]
RULE computeImplTrans

\[
\text{saveImplTrans}(\text{setUnion}(\text{setWrap}(\text{ITrans}), \text{setWrap}(\text{BaseItfITrans}))) \\
\cap \text{computeImplTrans} \left( \begin{array}{c}
\text{BaseItf:ClassType} \\
\text{Set}
\end{array} \right) \rightarrow_k \text{implTrans}
\]

\[
\langle \text{Class:ClassType} \rangle \text{crntClass} \\
\langle \text{Class} \rangle \text{classType} \\
\langle \text{ITrans:Set} \rangle \text{implTrans} \\
\langle \text{BaseItf} \rangle \text{classType}
\]

RULE computeImplTrans-Elem-Discard

computeImplTrans(\begin{array}{c}
\text{Set} \\
\text{Set}
\end{array} \rightarrow_k)

RULE computeImplTrans-Discard

computeImplTrans(\begin{array}{c}
\text{Set} \\
\text{Set}
\end{array} \rightarrow_k)

SYNTAX \ KItem ::= \text{saveImplTrans}(K) [\text{strict}]

RULE

\[
\text{saveImplTrans}(\text{setWrap}(S_1:Set)) \\
\rightarrow_k \langle \text{Class:ClassType} \rangle \text{crntClass} \\
\langle \text{Class} \rangle \text{classType}
\]

\[
\langle \text{S1} \rangle \text{implTrans}
\]

Inherits the methods of the base class, based on rules in JLS §8.4.8 paragraph 1

SYNTAX \ KItem ::= \text{tryInheritSet}(Set) \\
| \text{tryInherit}(\text{ClassType}) \\
| \text{tryInheritImpl}(\text{Map})

RULE tryInheritSet

\[
\text{tryInherit}(\text{Class:ClassType}) \\
\cap \text{tryInheritSet} \left( \begin{array}{c}
\text{Class} \\
\text{Set}
\end{array} \rightarrow_k \right)
\]

RULE tryInheritSet-Discard

\[
\text{tryInheritSet}(\begin{array}{c}
\text{Set} \\
\text{Set}
\end{array} \rightarrow_k)
\]

249
RULE 
\[
\text{tryInherit} (\text{Class}:\text{ClassType}) \quad \Rightarrow \quad \text{tryInheritImpl} (\text{Env}) \\
\] 
\[
\text{k} \quad \text{Class} \quad \text{classType} \quad \text{Env:Map} \quad \text{methods} 
\]

RULE tryInherit-Discard
\[
\text{tryInherit} (\cdot) \quad \Rightarrow \quad \cdot 
\]

RULE tryInheritImpl-Unfold
\[
\text{k} \quad \text{tryInheritImpl} (\text{MI}) \quad \Rightarrow \quad \text{tryInheritImpl} (\text{MI}, \text{MapItem}) \quad \text{-:MapItem} \quad \cdot 
\]

RULE tryInheritImpl
\[
\text{tryInheritImpl} (\text{Sig}:\text{K} \mapsto \text{DecClass}:\text{ClassType}) \\
\text{ifAux} (\text{isInheritable} (\text{getMethodAccessMode} (\text{methodRef} (\text{Sig}, \text{DecClass}))), \text{inherit} (\text{methodRef} (\text{Sig}, \text{DecClass})), \cdot) 
\]

RULE tryInheritImpl-empty-Discard
\[
\text{tryInheritImpl} (\cdot) \\
\text{\cdot} 
\]

SYNTAX 
\[
\text{KItem ::= isInheritable} (\text{AccessMode}) \quad \text{[strict]} 
\]

RULE isInheritable
\[
\text{isInheritable} (\text{BaseAccessMode}:\text{AccessMode}) \quad \Rightarrow \quad \text{isOverridden} (\text{BaseClass}, \text{BaseAccessMode}, \text{Class}) \\
\text{k} \quad \text{Class} \quad \text{classType} \quad \text{BaseClass}\text{.ClassType} \quad \text{extends} 
\]

SYNTAX 
\[
\text{KItem ::= isOverridden} (\text{ClassType}, \text{AccessMode}, \text{ClassType}) 
\]

RULE
\[
\text{isOverridden} (\cdot, \text{public}, \cdot) \quad \Rightarrow \quad \text{true} 
\]

RULE
\[
\text{isOverridden} (\cdot, \text{protected}, \cdot) \\
\text{true} 
\]
RULE

\[
\text{isOverridden (BaseC:ClassType, package, SubC:ClassType)}
\]

\[
eq\text{Aux ( getPackage ( getTopLevel (BaseC) ), getPackage ( getTopLevel (SubC) ) )}
\]

RULE

\[
\text{isOverridden (—, private, —)}
\]

\[
\text{false}
\]

Inherit a method by the current class. The inherited method overwrites previous methods with the same signature in <methods>.

SYNTAX

\[
K\text{Item} ::= \text{inherit (MethodRef)}
\]

RULE INHERIT

\[
\begin{array}{c}
\text{inherit (methodRef (Sig:Signature, DecClass:ClassType))} \\
\text{Env:Map [ DecClass / Sig ] methods}
\end{array}
\]

\[
\begin{array}{c}
\langle \text{Class} \rangle_{\text{classType}} \\
\langle \text{Class} \rangle_{\text{crntClass}}
\end{array}
\]

C.6.4 Method declarations

Methods are now typed and we need to store their types in their closures, so that their type contract can be checked at invocation time. The rule below is conceptually similar to that of untyped KOOL; the only difference is the addition of the types.

CONTEXT

\[
'M\text{ethodDec}'('M\text{ethodDecHead}(\text{—}:K, \text{—}:K, \square, \text{—}), \text{—})
\]

CONTEXT

\[
'M\text{ethodDec}'('M\text{ethodDecHead}(\text{—}:K, \text{—}:K, \text{—}:\text{Type}, \text{—}:\text{Id}, [ \square ), '\text{Param}(\text{—}:K, \square, \text{—}:K), \text{— }, \text{—}:\text{K}, \text{—}:\text{K})
\]

RULE MethodDec

\[
'M\text{ethodDec}'('M\text{ethodDecHead}('\text{Modifiers}:\text{KListWrap}, \text{—}:K, \text{ReturnType}:\text{Type}, \text{Name}:\text{Id}, ['\text{Params}:\text{KList}], \text{—}:K), \text{Body}:K)
\]

\[
\text{storeMethod ( [ Params ], getContentType (Modifiers), getAccessMode (Modifiers), isSynchronizedModifiers (Modifiers), methodMMT, \gamma, Body, sig (Name, getTypes ( [ Params ] )), ReturnType)
\]

\[
\langle \text{Class} : \text{ClassType} \rangle_{\text{crntClass}}
\]

REQUIRES \text{paramTypesResolved ( [ Params ] )}

\[\text{[structural]}\]
SYNTAX  \( KItem ::= \text{paramTypesResolved}( \text{KListWrap} ) \) [function]

RULE
\[
\text{paramTypesResolved} \left( \left[ \begin{array}{c}
\text{paramTypesResolved} \left( \left[ \begin{array}{c}
\text{'Param}(-:K, T::Type, -:Id) \\
\text{KList}
\end{array} \right]\right) \\
\text{KList}
\end{array} \right]\right)
\]

RULE
\[
\text{true}
\]

RULE  \text{AbstractMethodDec}
\[
\text{'AbstractMethodDec}(-:K, \text{Ks:KList}) \\
\text{'MethodDec}('\text{MethodDecHead}\left( \left[ \begin{array}{c}
\text{'Public}('\text{KList}), \text{'Abstract}('\text{KList}), \text{Ks}
\end{array} \right]\right), \text{'NoMethodBody}('\text{KList})
\]

SYNTAX  \( KItem ::= \text{storeMethod}( \text{KListWrap}, \text{ContextType}, \text{AccessMode}, \text{TypedVal}, \text{MethodMetaType}, K, K, \text{Signature}, \text{Type} ) \)

RULE  \text{storeMethod-Synchronized-method-inst-desugar}
\[
\text{true} \quad \text{false} \\
\text{Body:K} \\
\text{'Synchronized}('\text{This}('\text{KList}), \text{Body})
\]

RULE  \text{storeMethod-Synchronized-method-static-desugar}
\[
\text{true} \quad \text{false} \\
\text{Body:K} \\
\text{'Synchronized('Lit('\text{Class}(\text{Class})), \text{Body})}
\]
RULE storeMethod

\[
\text{storeMethod} \left( \begin{array}{c}
\text{Params: KList}, \text{CT: ContextType}, \text{Acc: AccessMode}, \text{false :: bool}, \\
\text{MMT: MethodMetaType}, \text{FirstLine: K}, \text{Body: K}, \text{Sig: Signature}, \text{ReturnType: Type}
\end{array} \right) \xrightarrow{k} 'K \\
\langle \text{Class: ClassType} \rangle \_crntClass \langle \text{Class} \rangle \_classType \langle \text{Env: Map} \rangle \_\text{methods}
\]

\[
\begin{array}{c}
\langle \text{Sig} \rangle \_\text{methodSignature} \langle \text{ReturnType} \rangle \_\text{methodReturnType} \\
\langle \text{FirstLine} \rangle \_\text{methodConstrFirstLine} \langle \text{Body} \rangle \_\text{methodBody} \langle \text{Acc} \rangle \_\text{methodAccessMode} \\
\langle \text{CT} \rangle \_\text{methodContextType} \langle \text{MMT} \rangle \_\text{methodMetaType}
\end{array}
\]

REQUIRES \neg \text{Bool} \left( \text{Sig in keys (Env)} \land \text{Bool MMT =}_K \text{constructorMMT} \right)

Discard the default constructor that was added by the rule \processTypeDecsInPCUPhase-ClassDec if this class already has a default constructor.

RULE storeMethod-discard-default-constructor

\[
\text{storeMethod} \left( \begin{array}{c}
\text{constructorMMT}, \text{Sig: Signature}
\end{array} \right) \xrightarrow{k} 'K \\
\langle \text{Class: ClassType} \rangle \_crntClass \langle \text{Class} \rangle \_classType \langle \text{Env [ Class / Sig ]} \rangle \_\text{methods}
\]

C.6.5 Constructor declarations

RULE ConstrDec-SuperCall-Desugar

\[
\text{ConstrDec} \left( \begin{array}{c}
\text{ConstrDecHead}(\text{superConstrInv}, \text{ConstrBody}(\text{None}(\text{params}))}, \text{ConstrBody}(\text{None}(\text{params}))
\end{array} \right) \xrightarrow{k} 'K \\
\langle \text{Class: ClassType} \rangle \_crntClass
\]

REQUIRES Class \neq_K \text{class Object}

RULE ConstrDec-Object-Desugar

\[
\text{ConstrDec} \left( \begin{array}{c}
\text{ConstrDecHead}(\text{params}), \text{ConstrBody}(\text{None}(\text{params}))}
\end{array} \right) \xrightarrow{k} 'K \\
\langle \text{Class: ClassType} \rangle \_crntClass
\]

REQUIRES Class =_K \text{class Object}

CONTEXT

\[
\text{ConstrDec} \left( \begin{array}{c}
\text{ConstrDecHead}(\text{params}), \text{ConstrBody}(\text{None}(\text{params}))}
\end{array} \right) \xrightarrow{k} 'K
\]

253
C.6.6 Instance fields and instance initializers

CONTEXT

\texttt{FieldDec(\text{\ldots} :K, \phi, \text{\ldots})}

RULE FieldDec-Multi-Desugar

\[
\frac{\text{\ldots}}{\text{\ldots} \triangleright FieldDec(\text{\ldots} :K, T, [ K1 \text{ \ldots} ])}
\]

RULE FieldDec-with-init-desugar

\[
\frac{\text{\ldots}}{\text{InstanceInit(ExprStm(Assign(ExprName(X), InitExp))))}
\]

REQUIRES (getKLabel(InitExp) \neq KLabel 'ArrayInit) \land \text{Bool (getContextType} (\text{Modifiers}) \\
= K \text { instance})

[structural]

RULE FieldDec-Instance

\[
\frac{\text{\ldots}}{\text{\ldots} \triangleright FieldDec(\text{\ldots} :K, T, [ \text{\ldots} ])}
\]

\[
\frac{\text{\ldots}}{\text{\ldots} \triangleright FieldDec([ \text{\ldots} ], T, [ \text{\ldots} ])}
\]

REQUIRES \text{getContextType} (\text{Modifiers}) = K \text { instance}

[structural]
rule InstanceInit
\[
\begin{align*}
\text{(InstanceInit)} (K,K) &\rightarrow^k \langle \text{Class:ClassType} \rangle_{\text{crntClass}} \langle \text{Class} \rangle_{\text{classType}} \\
\langle [-, 'K\text{List}] \rangle \text{ instancInit} &\rightarrow^k K
\end{align*}
\]
[structural]

SYNTAX \( K\text{ItemList} ::= \text{convertInstanceInitIntoMethod} \)

rule convertInstanceInitIntoMethod
\[
\text{convertInstanceInitIntoMethod} \rightarrow^k \text{storeMethod} ( [ 'KList ], \text{instance}, \text{private}, \text{false} :: \text{bool}, \text{methodMT}, 'K, 'Block ( [ InstanceInitKs ] ) ) , \text{sig} ( \text{String2Id} ( "$\text{instance\_init}$" ) , \text{types} ( 'KList ) ) , \text{void} )
\]
\[
\langle \text{Class:ClassType} \rangle_{\text{crntClass}} \langle \text{Class} \rangle_{\text{classType}} \langle [ 'InstanceInitKs:KList ] \rangle \text{ instancInit} \rightarrow^k \text{Bag}
\]

C.6.7 Static fields and static initializers

Desugaring a static field declaration with initializer into a declaration without initializer. Followed by a static initializer block, if this field is not a compile-time constant.

rule FieldDec-StaticInit-Desugar
\[
\text{FieldDec} ( \text{Modifiers:KListWrap}, \text{T:Type}, [ '\text{VarDec} (X, \text{InitExp:KList}) ] ) \leadsto \langle [-, 'K] \rangle \\
\text{StaticInit} ( '\text{ExprStm} ( '\text{Assign} ( '\text{ExprName} (X), \text{InitExp} ) ) )
\]

REQUIRES \( \langle \text{getKLabel} ( \text{InitExp} ) \neq \text{KLabel 'ArrayInit} \rangle \land \text{Bool ( getKLabel ( \text{InitExp} ) = K \text{static } ) } \land \text{Bool ( isFinalModifiers ( \text{Modifiers} ) \land Bool ( getKLabel ( \text{InitExp} ) = K \text{Lit } \lor \text{Bool isKResult (InitExp) = true } ) )} \)
[structural]

rule FieldDec-Static
\[
\text{FieldDec} ( \text{Modifiers:KListWrap}, \text{T:Type}, [ '\text{VarDec} (X, ) ] ) \rightarrow^k \langle \text{Class:ClassType} \rangle_{\text{crntClass}} \\
\langle [ -, 'K\text{List}] \rangle \text{ FieldDec ( [ 'Static ( 'KList ) ] , T, [ 'VarDec (X) ] )} \rightarrow^k \text{staticFields}
\]

REQUIRES getKLabel ( \text{Modifiers} ) = K \text{static}
[structural]
The parser represents interface fields as 'ConstantDec labels. This rule desugars them into 'FieldDec labels and adds the modifiers "public static final". Interface fields are not necessarily compile-time constants.

### C.6.8 Compile-time constants

**CONTEXT**

\[
\text{'FieldDec}(\text{Modifiers}:K, \ T:T, [ \text{'VarDec}(X:Id, \square) ]) \]

**REQUIRES**

\[ \text{getContextType}(\text{Modifiers}) = K\text{ static} \land \text{Bool isFinalModifiers}(\text{Modifiers}) \]
\[ \land \text{Bool getKLabel}(\square) = K\text{Label} 'Lit \]

**RULE FieldDec-compile-time-constant**

\[
\text{'FieldDec}(\text{Modifiers}:K\text{ListWrap}, \ T:T, [ \text{'VarDec}(X:Id, TV:TypedVal) ]) \]
\[ \text{constantEnv} \]

**REQUIRES**

\[ \text{getContextType}(\text{Modifiers}) = K\text{ static} \land \text{Bool isFinalModifiers}(\text{Modifiers}) \]

### C.6.9 Other members

Discard inner class declarations at this phase. They are processed when their respective <class> tag is encountered as part of processClasses.

**RULE ClassDec-discard**

\[
\text{'ClassDec}(--) \]

**RULE InterfaceDec-discard**

\[
\text{'InterfaceDec}(--) \]
C.6.10 Functions for accessing member modifiers

Evaluates to true::bool if synchronized is among modifiers, false otherwise

**SYNTAX**  

\[ KItem ::= \text{isSynchronizedModifiers} ( KListWrap ) \]  

**RULE**

\[
\text{isSynchronizedModifiers} ( [ ';Synchronized(-), - ] ) \\
\text{true :: bool}
\]

**RULE**

\[
\text{isSynchronizedModifiers} ( [ \text{KL}:\text{KLabel}(-) ] \text{',KList}, - ] ) \\
\text{requires KL} \neq \text{'Synchronized} \\
\text{true :: bool}
\]

**RULE**

\[
\text{isSynchronizedModifiers} ( [ ';KList ] ) \\
\text{false :: bool}
\]

Used to provide an approximate implementation of the distinction between static constant and non-constant fields.

**SYNTAX**  

\[ KItem ::= \text{isFinalModifiers} ( KListWrap ) \]  

**RULE**

\[
\text{isFinalModifiers} ( [ ';Final(-), - ] ) \\
\text{true}
\]

**RULE**

\[
\text{isFinalModifiers} ( [ \text{KL}:\text{KLabel}(-) ] \text{',KList}, - ] ) \\
\text{requires KL} \neq \text{'Final} \\
\text{false :: bool}
\]
C.7 Module CORE-PREPROCESSING

Most commonly used auxiliary functions related to preprocessing semantics only. This module is designed to be included in most other modules of preprocessing.

C.7.1 Class-related functions

Returns a setWrap(\{ClassType\}), containing all top level classes in the program. Uses <namesToClasses> to compute the result. Used by the starting rule of several preprocessing phases.

**SYNTAX**

\[
KItem ::= \text{getTopLevelClasses} \\
| \text{getTopLevelClasses} (\text{Map}, \text{Set})
\]

**RULE getTopLevelClasses-start**

\[
\text{getTopLevelClasses} \quad \text{getTopLevelClasses} (\text{NameMap}, \text{Set}) \quad \text{k} \quad \text{<namesToClasses>}
\]

**RULE getTopLevelClasses-top-level**

\[
\text{getTopLevelClasses} (\quad \mapWrap (\quad \text{Class:ClassType}, \quad \text{\text{'Set}}) \quad \text{k} \quad \text{Class}) \\
\langle \text{Class} \rangle \text{classType} \quad \{\text{enclosingClass}\}
\]

**RULE getTopLevelClasses-not-top-level**

\[
\text{getTopLevelClasses} (\quad \mapWrap (\quad \text{Class:ClassType}, \quad \text{class}) \quad \text{k}) \\
\langle \text{Class} \rangle \text{classType} \quad \{\text{class} \quad \text{enclosingClass}\}
\]

**RULE**

\[
\text{getTopLevelClasses} (\quad \mapWrap (\quad \text{\text{'Map}}), \quad \text{)}
\]

**RULE**

\[
\text{getTopLevelClasses} (\quad \text{'Map}, \quad \text{ClassesSet:Set}) \\
\text{setWrap (ClassesSet)}
\]

Returns a setWrap(\{ClassType\}), containing all direct inner classes of the given class. Uses <namesToClasses> to compute the result.

**SYNTAX**

\[
KItem ::= \text{getInnerClasses} (\text{ClassType}) \\
| \text{getInnerClasses} (\text{Map}, \text{Set})
\]
C.7.2 Method-related functions

Returns the access mode of a method closure given as argument

SYNTAX  \( KItem ::= \text{getMethodAccessMode} ( \text{MethodRef} ) \) \[\text{strict}\]

RULE \text{getMethodAccessMode}

\[
\begin{align*}
\text{getMethodAccessMode} & ( \text{methodRef} ( \text{Sig} , \text{Class}) ) \rightarrow \text{ Acc } \\
\text{methodSignature} & \rightarrow \text{Acc:AccessMode} \\
\text{methodAccessMode} & \rightarrow \text{classType}
\end{align*}
\]

SYNTAX  \( KItem ::= \text{getContextType} ( \text{KListWrap} ) \) \[\text{function}\]

RULE \text{getContextType}

\[
\begin{align*}
\text{getContextType} & ( [ \text{'Static'(-)}, \text{---} ] ) \\
\text{static} & \rightarrow \\
\end{align*}
\]
RULE
\[
\text{getContextType} \left( [ \text{KL:KLabel}(-), \cdots ] \right)
\]

REQUIRES KL \neq KLabel 'Static

RULE
\[
\text{getContextType} \left( [ 'KList ] \right)
\]

instance

Extracts the access mode from the list of modifiers of some Java entity.

SYNTAX \hspace{1em} KItem ::= \text{getAccessMode} \left( KListWrap \right) [\text{function}]

RULE
\[
\text{getAccessMode} \left( [ 'Public(-), \cdots ] \right)
\]

public

RULE
\[
\text{getAccessMode} \left( [ 'Protected(-), \cdots ] \right)
\]

protected

RULE
\[
\text{getAccessMode} \left( [ 'Private(-), \cdots ] \right)
\]

private

RULE
\[
\text{getAccessMode} \left( [ \text{KL:KLabel}(-), \cdots ] \right)
\]

REQUIRES \hspace{1em} (KL \neq KLabel 'Public) \land (KL \neq KLabel 'Protected) \land (KL \neq KLabel 'Private)

RULE
\[
\text{getAccessMode} \left( [ 'KList ] \right)
\]

package
C.8 Module ELABORATION-CORE

C.8.1 Elaboration phase — introduction

In this phase we elaborate the content of code blocks inside a class — the last step of preprocessing (besides Folding). During this phase we inspect the contents of method bodies, instance initializers and static initializers of a class and perform the following transformations:

- each variable name \(x\) is resolved into either:
  - \(x\) — a local var
  - \(\text{Class}.x\) — a static var defined in the class \(\text{Class}\)
  - \(\text{field(obj, Class, x)}\) — a field of object \(\text{obj}\) declared in the class \(\text{Class}\). Term \(\text{obj}\) could also be ‘This.

- each method name is resolved into either:
  - \(\text{Class}.m\) — a static method defined in the class \(\text{Class}\)
  - \(\text{method(obj, Class, x)}\) — an instance method of object \(\text{obj}\) declared in the class \(\text{Class}\). Term \(\text{obj}\) could also be ‘This. The actual version of the method will be looked up at runtime.

- each method signature is resolved into its appropriate overloaded version. To denote the version, each actual parameter will be casted to the type of the actual parameter.

- each expression \(\text{Exp}\) will be replaced with a corresponding typed expression \(\text{cast}(T, \text{Exp})\), \(T\) being the compile-time type of the expression.

During elaboration, elaborated members will be wrapped into \(\text{elab()}\).

During elaboration an expression transition through 6 phases:

1. Elaboration heating: \(\text{elab('CastRef(_, 'Minus('ExprName(x))))} \Rightarrow \text{elab('Minus('ExprName(x)))} \Rightarrow \text{elab('CastRef(_, HOLE)}\) Some AST terms, especially some statements require custom elaboration heating rules. If the heated expression should be always computed into a KResult, such as a type, package or certain literal expresssions, then it is heated in the “naked” form, e.g. not wrapped into \(\text{elab()}\).

2. Elaboration of children. All the children of the expression are elaborated. After this phase elaborated children will be typed (if they are expressions) and wrapped into \(\text{elabRes()}\). Typed expressions are cast expressions - like \(\text{cast}(T, \text{Exp})\). The whole initial term will be converted into:
   \(\text{elab('Minus(\text{elabRes(cast(int, localVar(X))))}) \Rightarrow \text{elab(\text{CastRef(_, HOLE)}}\)

3. Initiation of the step elabDispose. When all children have been elaborated and are either KResult of \(\text{elabRes()}\), the wrapper is changed from \(\text{elab()}\) to \(\text{elabDispose()}\). elabDispose('Minus(\text{elabRes(cast(int, localVar(X))))}) \Rightarrow \text{elab(\text{CastRef(_, HOLE)}}\)

4. Unwrapping of children. During elabDispose step elaborated children are unwrapped from their \(\text{elabRes()}\) wrapper. elabDispose('Minus(cast(int, localVar(X)))) \Rightarrow \text{elab(\text{CastRef(_, HOLE)}}\)
5. End of the step elabDispose. When all children of the term wrapped into elabDispose have been unwrapped from their elabRes() wrapper, the root wrapper is replaced from elabDispose to elabEnd. This contributes to more simple rules for the following steps. elabRes('Minus(cast(int, localVar(X)))) ~> elab(CastRef(_, HOLE)

6. Computation of elaboration result. Now that all children have been elaborated and unwrapped, it is possible to compute the type of the current expression itself. When the expression is fully elaborated, it is wrapped into elabRes(). This is the step that requires custom rules for most AST terms.

   elabRes('Minus(cast(int, cast(int, localVar(X)))))) ~> elab(CastRef(_, HOLE)

7. Elaboration cooling. Once the top K Item was wrapped into elabRes, it is ready to be cooled back into its original context: elab('CastRef(_, elabRes('Minus(cast(int, cast(int, localVar(X))))))))

Module ELABORATION-CORE contains the backbone of elaboration phase.

C.8.2 Core auxiliary definitions

Custom hole used for custom heating/cooling rules in the elaboration phase.

**SYNTAX**

\[ KItem ::= \text{CHOLE} \]

Elaborates the given statement/expression. The first step of elaboration.

**SYNTAX**

\[ KItem ::= \text{elab}(K) \]

Wraps the elaboration result. Since elaboration may happen at both ElaborationPhase and Execution-Phase, it cannot be KResult. Actually it is not KResult for HOLE, but is for CHOLE.

**SYNTAX**

\[ ElabKResult ::= \text{elabRes}(K) \]

**RULE**

\[ \text{isElabKResult(KResult)} \]

<table>
<thead>
<tr>
<th>KR: KResult</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
</tbody>
</table>

C.8.3 Elaboration of code blocks

Heating arguments for both expression and statement terms. The attribute [transition-strictness] is used as transition attribute for testing strictness. This is a rule that may lead to unexpected nondeterminism if it is wrongly implemented. In order to expose incorrect nondeterminism we need to model-check a program that exposes the nondeterminism.

Labels that are 'naked' should be heated in their pure form. Other terms should be heated wrapped into elab().
RULE ELAB-HEAT-DEFAULT

\[
\begin{align*}
\not\equiv \rule{Elab}{\text{elab} (\mathbf{K}:\text{KLabel}(\mathbf{KL}:\mathbf{KLabel} ( \mathbf{KL}:\mathbf{K})))} \\
\text{requires } \left( \text{defaultElabHeating} (\mathbf{KL}) = \mathbf{K} \text{ true } \lor \text{customElabHeating} (\mathbf{KL}, [\text{Headks}, \text{K}]) = \mathbf{K} \text{ true } \right) \\
\land \text{Bool} \neg \text{isElabKResult} (\mathbf{K}) = \mathbf{K} \text{ true } \\
\end{align*}
\]

RULE ELAB-COOL-DEFAULT

\[
\begin{align*}
\not\equiv \rule{Elab}{\text{elab} (\mathbf{K}:\text{KLabel}(\mathbf{CL}:\text{KLabel}(\mathbf{K}:\mathbf{K})))} \\
\text{requires } \text{isElabKList} ([\mathbf{ElabResL}]) = \mathbf{K} \text{ true } \\
\end{align*}
\]

Must be true for AST nodes that require elaboration heating of their children. For the arguments KL, Ks, the original node is of the form KL(KS, \_\) and the heated child is always the last element of Ks. Thus it is possible to decide whether the child is allowed to be heated based both on the position of the child in the list of arguments as well as based on the state of previous brothers.

This predicate is intended to be used for KLabels that require specific child heating rules based on the state of their children. The KLabels that heat all their children using default rules are enumerated in the predicate defaultElabHeating.

The predicate definition is scattered across modules related to elaboration.

SYNTAX \ KItem ::= \text{customElabHeating} (\text{KLabel}, \text{KListWrap}) [\text{function}]

The default algorithm of transforming the term from elab to elabRes, when the children were completely elaborated. Deletes elabRes wrappers from children. This algorithm is activated when the following conditions apply:

- term is not customElabChildren
- term children are completely elaborated — isElab(children)
- term is not naked. This case should never be true, but there is some weird case that requires it.

When the default algorithm is not appropriate, the respective term should be in the category customElabChildren

SYNTAX \ KItem ::= \text{elabDispose} (\ mathbf{K})

This rule is universal for all forms of terms, both instructions and statements.

RULE ELAB-TO-ELABDISPOSE

\[
\begin{align*}
\not\equiv \rule{Elab}{\text{elab} (\mathbf{KL}:\text{KLabel} (\mathbf{ElabResL}:\text{KList}))} \\
\text{elabDispose} (\mathbf{KL} (\mathbf{ElabResL})) \\
\text{requires } \text{isElabKList} ([\mathbf{ElabResL}]) = \mathbf{K} \text{ true } \\
\end{align*}
\]
RULE elabDispose-process

\[
\text{elabDispose (KL, KLLabel (_, elabRes (K:K), _))}
\]

RULE elabDispose-to-elabEnd

\[
\text{elabDispose (KL, KLLabel (Ks:KList))}
\]
\[
\text{elabEnd (KL (Ks))}
\]

REQUIRES \(\neg \text{Bool haveElabRes ([Ks] = K true)}\)

True if given KList has no terms of the form \(\text{elabRes(...)}\), false otherwise.

SYNTAX \(KItem ::= \text{haveElabRes (KListWrap)} \) [function]

RULE

\[
\text{haveElabRes ([_, elabRes (_), _])}
\]
true

RULE

\[
\text{haveElabRes ([_, KList2KLabel _, elabRes (_), _, _, _])}
\]
true

The 3rd elaboration-phase wrapper for expressions. Represents the case when children are completely elaborated and unwrapped from \(\text{elabRes()}\), but root node might not be elaborated yet. The implementation is scattered across all modules that deal with the elaboration phase.

SYNTAX \(KItem ::= \text{elabEnd (K)}\)

C.8.4 Auxiliary functions for elaboration phase

Computes into an expression of the form \(\text{elabRes(cast(QualClass, 'QThis(QualClass)))}\), where QualClass is searched in the enclosing context of the first argument, being a subclass of the second one. Or noValue if no suitable result is found.

SYNTAX \(KItem ::= \text{findQualifierOfType (ClassType, ClassType)} \) [strict(2)]

RULE

\[
\text{isElabNaked (findQualifierOfType (_ _, _ _))}
\]
true
RULE elabEnd-findQualifierOfType

\[
\text{findQualifierOfType}(\text{QualClass}:\text{ClassType}, \text{ReqClass}:\text{ClassType})
\]

\[
\text{ifAux}(\text{subtype}(\text{QualClass}, \text{ReqClass}), \text{elabRes}(\text{cast}(\text{QualClass}, 'Q\text{This}'\ldots))^{k}, \text{findQualifierOfType}(\text{QualEncloserClass}, \text{ReqClass}))
\]

\[
\text{findQualifierOfType}(\text{QualClass}:\text{classType}, \text{QualEncloserClass}:\text{ClassType})^{\text{enclosingClass}}
\]

REQUIRES \text{ReqClass} \neq \text{K}^{K}\ldots$

RULE elabEnd-findQualifierOfType-top-level

\[
\text{findQualifierOfType}(\text{K}:\text{K}, \text{K}')^{K}
\]

Happens for 'NewInstance expressions for static inner classes.

RULE elabEnd-findQualifierOfType-static

\[
\text{findQualifierOfType}(\text{K}', \text{K}:\text{K})^{K}
\]

Computes to true if the given argument is a list of elaboration results, false otherwise. An elaborated result is either:

- KResult
- \text{elabRes}(\ldots)

It is notably NOT a \text{cast()} expression. This, together with the fact that \text{cast()} has no elaboration heatig rules, makes the whole elaboration phase non-reentrant.

SYNTAX \quad \text{KItem ::= isElabKList( } \text{KListWrap } \text{)[function]}$

RULE

\[
\text{isElabKList( } \lbrack \text{ElabKR:ElabKResult, Ks:KList} \rbrack \text{)}^{K}
\]

isElabKList( \lbrack Ks \rbrack)

RULE

\[
\text{isElabKList( } \lbrack \text{KList} \rbrack\ldots)
\]

true

C.8.5 Elaboration of statements — the step elabEnd

Applies to all nodes that are not expressions — statements and non-expression auxiliary functions
RULE ELABEND-DEFAULT-ALL-EXCEPT-EXPRESSIONS

\[ \text{elabEnd} ( \text{KL:KLabel(} Ks:KList) ) \]
\[ \text{elabRes} ( \text{KL(Ks)} ) \]

REQUIRES \( \neg \text{Bool} ( \text{isExp(KL(Ks))} =_K \text{true} \lor \text{customElabEnd (KL)} =_K \text{true} ) \)

The implementation is scattered across elaboration-related modules

SYNTAX \[ KItem ::= \text{customElabEnd (KLabel)} \ [\text{function}] \]

Eliminate stmtAndExp if the first argument is empty, otherwise keep it for execution phase.

RULE

\[ \text{elabEnd ( stmtAndExp (Stmt:K, cast (T:Type, Exp:K)) )} \]
\[ \# \text{if Stmt} \neq_K \# \text{then elabRes (cast (T, stmtAndExp (Stmt, cast (T:Type, Exp:K))) )} \]
\[ \# \text{else elabRes (cast (T:Type, Exp:K)) } \# \text{fi} \]

C.8.6 Elaboration of KListWrap — the list of terms

RULE ELAB-HEAT-KLISTWRAP

\[ \ineq_K \hookrightarrow \text{elab (} [ \text{HeadKs:KList, } \text{K:KCHOLE}, \text{KList}] ) \]

REQUIRES \( \text{isElabKList ( [ HeadKs ] )} \land \text{Boot} \neg \text{Boot \ isElabKResult(K) } =_K \text{true} \)

RULE ELAB-COOL-KLISTWRAP

\[ \text{ElabK:ElabKResult} \hookrightarrow \text{elab (} [ \text{--}, \text{CHOLE}, \text{--} ] \) \]

RULE ELAB-TO-ELABDISPOSE-KLISTWRAP

\[ \text{elab (} [ K1:K, \text{ElabResL:KList} ] \) \]
\[ \text{elabDispose (} [ K1, \text{ElabResL} ] \) \]

REQUIRES \( \text{isElabKList (} [ K1, \text{ElabResL} ] \) =_K \text{true} \)

RULE ELABDISPOSE-PROCESS-KLISTWRAP

\[ \text{elabDispose (} [ \text{--}, \text{elabRes (K:K)} ] \)
\[ K \]
RULE elabDispose-to-elabEnd-KListWrap

\[
\text{elabDispose} \left( \left[ K1.K, Ks.KList \right] \right) \\
\text{elabRes} \left( \left[ K1, Ks \right] \right)
\]

REQUIRES \( \neg \text{Bool} \text{haveElabRes} \left( \left[ Ks \right] \right) = \text{true} \)

RULE elab-to-elabRes-KListWrap-empty

\[
\text{elab} \left( \left[ \cdot.KList \right] \right) \\
\text{elabRes} \left( \left[ \cdot.KList \right] \right)
\]

C.8.7 Auxiliary functions for other modules

Removes the last layer from \(<\text{elabEnv}>\)

SYNTAX \( KItem ::= \text{removeLastElabEnv} \)

RULE removeLastElabEnv

\[
\text{removeLastElabEnv} \left( \left[ K \right], \left[ \cdot.KList \right], \left[ \cdot.KList \right], \left[ \cdot.localTypes \right] \right)
\]

Operator ??

Chain of responsibility pattern. Evaluate the first argument. If it is KResult (except noValue) or \(\text{elabRes}()\), the result of the ?? expression is the result of the first argument. Otherwise, if the first argument evaluates to noValue, the result of the ?? expression is the result of the second argument.

SYNTAX \( KItem ::= K \ ?? \ K \)

RULE chainOfResponsibility-Heat

\[
\text{Arg1} \rightarrow \text{Arg1} : \text{K} \rightarrow \text{CHOLE} \quad ?? \quad \text{arg}
\]

REQUIRES \( \neg \text{Bool} \text{isElabKResult}(\text{Arg1}) = \text{true} \)

RULE chainOfResponsibility-Result1

\[
\text{ElabRes} : \text{ElabKResult} \rightarrow \left( \text{CHOLE} \ ?? \ \cdot \right)
\]

REQUIRES \( \text{ElabRes} \neq \text{K} \ ? \ K \)

RULE chainOfResponsibility-Result2

\[
\text{K} \rightarrow \text{K} : \text{K} \rightarrow \text{K}
\]
C.9 Module ELABORATION-TOP-BLOCKS

This module initiates the elaboration phase. It is responsible for elaborating all top-level code blocks in the program: methods, constructors, static and instance initializers.

**rule ElaborationPhase-start**

\[
\text{elaborateBlocks}(\text{getTopLevelClasses}) \xrightarrow{\text{ProcClassMembersPhase}} \text{ElaborationPhase} \xrightarrow{\text{globalPhase}}
\]

Elaborate the blocks inside all classes. Argument \( K = \text{setWrap}(\text{Set}) \) — the set of all classes.

**syntax**

\[ KItem ::= \text{elaborateBlocks}(K) \quad \text{[strict]} \]

It is important to elaborate the instance initializers before the methods. This way, when 'SuperConstrEnv is encountered, it inserts the already elaborated instance initializer in its place, avoiding name collisions between constructor arguments and fields inside instance init.

**rule elaborateBlocks**

\[
\begin{align*}
\text{elabMethods}(\text{Methods}) & \xleftarrow{\text{elabStaticInit}} \xleftarrow{\text{elaborateBlocks}(\text{getInnerClasses}(\text{Class}))} \xleftarrow{\text{restoreCrntClass}(\text{OldCrntClass})} \xleftarrow{\text{elaborateBlocks}(\text{setWrap}(\text{Class}:\text{ClassType} \rightarrow \text{Set}))} \\
& \xrightarrow{\text{oldCrntClass}(\text{Class})} \langle\langle\langle \text{Methods}:\text{Map} \rangle\rangle\rangle \text{methods}
\end{align*}
\]

**rule elaborateBlocks-discard**

\[ \text{elaborateBlocks}(\text{setWrap}(\text{Set})) \]

Elaborates the methods of the current class. The map contains the methods that were not elaborated yet. When a class elaboration starts, the map is initialized with the content of \(<\text{methods}>\) — all the methods accessible in the current class. Only methods declared in the current class are elaborated. The rest are discarded from the map.

**syntax**

\[ KItem ::= \text{elabMethods}(\text{Map}) \]
rule elabMethods-Heat-MethodParams

\[ \langle \text{addElabEnv} \cdot \text{elab} (\text{ParamsList}) \rangle \leadsto \text{elabMethods} (\langle \text{Sig} : \text{Signature} \mapsto \text{Class} \rangle_{\text{Map}} \rangle \sim k \] 

\[ \langle \text{Class} : \text{ClassType} \rangle_{\text{cntClass}} \leadsto \langle \text{contextType} \rangle_{\text{classType}} \langle \text{Sig} \rangle_{\text{methodSignature}} \] 

\[ \langle \text{ParamsList} : K \rangle_{\text{CHOLE}} \langle \text{methodParams} \rangle_{\text{contextType}} \langle \text{Class} : \text{ContextType} \rangle_{\text{methodContextType}} \] 

rule elabMethods-Heat-MethodFirstLine

\[ \langle \text{elabRes} (\text{ParamsList} : K) \rangle \leadsto \text{elab} (\langle \text{FirstLine} \rangle) \leadsto \text{elabMethods} (\langle \text{Sig} : \text{Map} \rangle \sim k \langle \text{Class} : \text{ClassType} \rangle_{\text{cntClass}} \] 

\[ \langle \text{contextType} \rangle_{\text{classType}} \langle \text{Class} \rangle_{\text{methodParams}} \langle \text{ParamsList} \rangle_{\text{methodContextType}} \] 

\[ \langle \text{FirstLine} : K \rangle_{\text{CHOLE}} \langle \text{methodConstrFirstLine} \rangle_{\text{classType}} \langle \text{CT} : \text{ContextType} \rangle_{\text{methodContextType}} \] 

Required when processing first line of the constructor of Object, which is .K

rule elab-DotK

\[ \text{elab} (\langle k \rangle) \] 

\[ \text{elabRes} (\langle k \rangle) \] 

rule elabMethods-Heat-MethodBody

\[ \langle \text{elabRes} (\text{FirstLine} : K) \rangle \leadsto \text{elab} (\langle \text{Body} \rangle) \leadsto \text{elabMethods} (\langle \text{Sig} : \text{Map} \rangle \sim k \langle \text{Class} : \text{ClassType} \rangle_{\text{cntClass}} \] 

\[ \langle \text{Class} \rangle_{\text{classType}} \langle \text{CHOLE} \rangle_{\text{methodConstrFirstLine}} \langle \text{Body} : K \rangle_{\text{CHOLE}} \langle \text{methodBody} \rangle \] 

rule elabMethods-Cool-Method

\[ \langle \text{elabRes} (\text{Body} : K) \rangle \leadsto \text{removeLastElabEnv} \text{elab} (\langle \text{Body} \rangle) \leadsto \text{elabMethods} (\langle \text{Sig} : \text{Map} \rangle \sim k \langle \text{Class} : \text{ClassType} \rangle_{\text{cntClass}} \] 

\[ \langle \text{Class} \rangle_{\text{classType}} \langle \text{methodConstrFirstLine} \rangle_{\text{methodBody}} \langle \text{Body} : K \rangle_{\text{methodMetaType}} \]
RULE `elabMethods-Cool-Constructor`
\[
\begin{align*}
\text{elabRes} \ (\text{Body}:K) & \quad \leadsto \quad \text{elabMethods} \ (\text{Map} : \text{Map}) \quad \leadsto \quad k \quad \langle \text{Class}:\text{ClassType} \rangle \text{crntClass} \\
\text{removeLastElabEnv} & \\
\text{Class} \langle \text{classType} \rangle \ (\text{FirstLine}:K) & \quad \text{methodConstrFirstLine} \quad \leadsto \quad \text{ CHOLE } \ [ \text{FirstLine}, \text{Body} ] \quad \text{methodBody} \\
\text{constructorMMT} & \\
\text{methodMMT} & \\
\text{methodMetaType} & \\
\text{constructorMMT} & \\
\text{methodMMT} & \\
\text{methodMetaType} &
\end{align*}
\]

RULE `elabMethods-Discard-METHOD`
\[
\begin{align*}
\text{elabMethods} \ (\text{Map} : \text{Map}) & \quad \leadsto \quad k \quad \langle \text{Class}:\text{ClassType} \rangle \text{crntClass} \\
\text{requires} \quad \text{DeclClass} \neq _K \text{Class}
\end{align*}
\]

RULE `elabMethods-End`
\[
\begin{align*}
\text{elabMethods} \ (\text{Map}) & \quad \leadsto \quad _K
\end{align*}
\]

SYNTAX \ KItem ::= elabStaticInit

RULE `elabStaticInit-Heat`
\[
\begin{align*}
\text{addElabEnv} & \quad \text{elab} \ (K) \quad \leadsto \quad k \quad \langle \text{Class}:\text{ClassType} \rangle \text{crntClass} \\
\text{Class} \langle \text{classType} \rangle & \quad \text{K/K} \quad \text{staticInit} \quad \text{CHOLE} \quad \text{contextType} \\
\text{requires} \quad K \neq _K \text{CHOLE}
\end{align*}
\]

RULE `elabStaticInit-End`
\[
\begin{align*}
\text{elabRes} \ (K:K) & \quad \text{elabStaticInit} \quad \leadsto \quad k \quad \langle \text{Class}:\text{ClassType} \rangle \text{crntClass} \quad \langle \text{Class} : \text{classType} \rangle \\
\text{removeLastElabEnv} & \\
\text{CHOLE} & \\
K & \quad \text{staticInit} \\
\end{align*}
\]

Adds a new empty layer to \langle elabEnv \rangle

SYNTAX \ KItem ::= addElabEnv
C.10 Module ELABORATION-STATEMENTS

Only statements that need custom treatment during elaboration are included here. Most statements are elaborated by generic rules.

C.10.1 Statements for, block and catch

Elaboration of 'KListWrap, 'Block, 'For or 'Catch — nodes that may contain variable declarations.

SYNTAX  $KItem ::= isVarDecHolderLabel ( KLabel )$ [function]

RULE

$\text{isVarDecHolderLabel ( 'Block)}$

\[
\begin{array}{c}
\text{true}
\end{array}
\]

RULE

$\text{isVarDecHolderLabel ( 'For)}$

\[
\begin{array}{c}
\text{true}
\end{array}
\]

RULE

$\text{isVarDecHolderLabel ( 'Catch)}$

\[
\begin{array}{c}
\text{true}
\end{array}
\]

In addition to what default elab heating does, when we heat the first argument of 'Block, 'For or 'Catch we need to save a copy of elaboration environment, e.g. the last entry of <elabEnv> <localTypes>. This is because these statements are blocks that might declare local variables.

RULE elab-BLOCK-FOR-CATCH-HEAT-FIRSTSUBTERM

\[
\begin{array}{c}
\text{elab ( KL:KLabel(K:K, Ks:KList) )} \\
\text{elab ( K ) \rightarrow elab ( KL(CHOLE, Ks:KList) )}
\end{array}
\]

\[
\begin{array}{c}
\text{List} \\
\text{ElabEnvLI:ListItem -} \\
\text{elabEnv}
\end{array}
\]

\[
\begin{array}{c}
\text{List} \\
\text{LocalTypesLI:ListItem -} \\
\text{localTypes}
\end{array}
\]

REQUIRES $\text{isVarDecHolderLabel ( KL ) \equiv true \land \text{isElabKResult ( K ) \equiv true}$
C.10.2 Elaboration of catch parameters

Elaborate parameter type, if not elaborated yet. For catch clause.

\textbf{CONTEXT}\
\begin{align*}
\text{elab} ('\text{Param}(---:K, \square, ---:Id))
\end{align*}

Adds params to the <elabEnv>. Used in both ELABORATION-TOP-BLOCKS and ELABORATION-BLOCKS

\textbf{RULE elab-Param}\
\begin{align*}
\langle \begin{array}{c}
\text{elab} ('\text{Param}(K1:K, T:\text{Type}, X:Id))\\
\text{elabRes} ('\text{Param}(K1:K, T:\text{Type}, X:Id))
\end{array} \rangle \mapsto_k \langle \begin{array}{c}
\text{MapWrap} ('\text{Map}\\
X \mapsto T)\\
\text{elabEnv}
\end{array} \rangle
\end{align*}

C.10.3 Local variable declarations

\textbf{RULE elab-LocalVarDecStm-desugar}\
\begin{align*}
\text{elab} ('\text{LocalVarDecStm}'('\text{LocalVarDec}(Ks:KList)'))
\end{align*}

[structural]

Resolve the local var type, required to register the var in <elabEnv>

\textbf{CONTEXT}\
\begin{align*}
\text{elab} ('\text{LocalVarDec}(---:K, \square, ---:K))
\end{align*}
**RULE elab-LocalVarDec-multi-desugar**

elab ( 'LocalVarDec(K : K, T : Type, [ Var1 : K, Var2 : K, VarDecs : KList ])
[ 'LocalVarDec(K, T, [ Var1 ]), 'LocalVarDec(K, T, [ Var2, VarDecs ])]
[structural]

**RULE elab-LocalVarDec-with-init-desugar**

elab ( 'LocalVarDec(K : K, T : Type, [ 'VarDec(X : Id, InitExp : K) ])
[ 'LocalVarDec(K, T, [ 'VarDec(X : Id) ]), 'ExprStm(Assign('ExprName(X), InitExp))]
)

REQUIRES getKLabel(InitExp) ≠ KLabel 'ArrayInit
[structural]

**RULE elab-LocalVarDec**

elab ('LocalVarDec(K : K, T : Type, [ 'VarDec(X : Id) ]))
elabRes ('LocalVarDec(K, T, [ 'VarDec(X) ]))

mapWrap ('Map
X → T "elabEnv"elabRes ('LocalVarDec(K, T, [ 'VarDec(X) ]))

C.10.4 Explicit constructor invocations — this(), super(), A.super()

Desugaring unqualified superclass constructor invocation into a qualified one

**RULE elab-SuperConstrInv-desugar**

elab ( 'SuperConstrInv(K : K, [ Args : KList ])
[ 'QSuperConstrInv(if CT = K instance #then findQualifierOfType(EnclosingClass, BaseEnclosingClass) #else #fi, K, [ Args ])]

Class : ClassType
< crntClass
( Class )classType ( BaseClass : ClassType )extends ( EnclosingClass : ClassType )enclosingClass class
( BaseClass )classType ( BaseEnclosingClass : ClassType )enclosingClass class
( CT : ContextType )classContextType class

RULE

customElabEnd ('QSuperConstrInv)
true

273
RULE elabEnd-QSuperConstrInv-local-augment-arguments

`K`  \sim elabEnd ('QSuperConstrInv("...")

localClassGetExtraArgs (BaseClass)
Qual:K, Arg2.K, [ Args:KList ])

(Class:ClassType) \crntClass (Class)classType (BaseClass:ClassType)extends

QSuperConstrInv lookup is always unqualified

RULE elabEnd-QSuperConstrInv-to-lookupMethod

elabRes ([ ExtraArgs:KList ])

lookupMethod (BaseClass, true, getConsName (BaseClass), [ Args, ExtraArgs ])

\sim elabEnd ('QSuperConstrInv(Qual:K, Arg2.K, [ Args:KList ]))

(Class:ClassType) \crntClass (Class)classType (BaseClass:ClassType)extends

RULE elabEnd-QSuperConstrInv

methodLookupResult (__, [ ExpectedArgExps:KList ], instance ) \sim elabEnd ( 'QSuperConstrInv(Qual:K, Arg2:K, [ -- ]) )

elabRes ('QSuperConstrInv(Qual, Arg2, [ ExpectedArgExps ]))

RULE

customElabEnd ('AltConstrInv)

true

RULE elabEnd-AltConstrInv-local-augment-arguments

`K`  \sim elabEnd ('AltConstrInv("...")

localClassGetExtraArgs (Class)
Arg1:K, [ Args:KList ])

(Class:ClassType) \crntClass

AltConstrInv lookup is always unqualified

RULE elabEnd-AltConstrInv-to-lookupMethod

elabRes ([ ExtraArgs:KList ])

lookupMethod (Class, true, getConsName (Class), [ Args, ExtraArgs ])

\sim elabEnd ('AltConstrInv(Arg1:K, [ Args:KList ]))

(Class:ClassType) \crntClass

RULE elab-AltConstrInv

methodLookupResult (__, [ ExpectedArgExps:KList ], instance ) \sim elabEnd ( 'AltConstrInv(Arg1:K, [ -- ]) )

elabRes ('AltConstrInv(Arg1:K, [ ExpectedArgExps ]))
C.11 Module ELABORATION-TYPES

RULE

′ClassOrInterfaceType(TypeK:K, —)

TypeK

[structural]

RULE

′InterfaceType(TypeK:K, —)

TypeK

[structural]

RULE

′ClassType(TypeK:K, —)

TypeK

[structural]

Resolving fully qualified type names. A name like pack.p2.A is represented as: `TypeName(PackageOrTypeName(PackageOrTypeName(pack),p2),A)

RULE

′PackageOrTypeName(KRs:KList, K:K)

′TypeName(KRs, K) ?? ′PackageName([KRs, K])

REQUIRES isKResult(KRs)

[structural]

When we search for a class qualified by another class, we simply convert the qualifier into a package.

RULE TypeName-QualifiedClass

′TypeName(ClassQ:ClassType, X:Id)

′TypeName(toPackage(ClassQ), X)

[structural]

RULE TypeName-QualifiedPackage

′TypeName(Pack:PackageId, X:Id)

typeNameQualifiedImpl(getNamesMap(Pack), X)

[structural]

Retrieves the ClassType for the given names map and simple class name
SYNTAX  $KItem ::= \text{typeNameQualifiedImpl} (K, Id) \, [\text{strict}(1)]$

RULE typeNamedImpl-Found

$\text{typeNameQualifiedImpl} (\text{mapWrap} (X \mapsto \text{Class}:\text{ClassType} \rightarrow), X:Id)$

RULE typeNamedImpl-NotFound

$\text{typeNameQualifiedImpl} (\text{mapWrap} (\text{NamesMap}:\text{Map}), X:Id)$

REQUIRES $\neg \text{Bool} \, X \, \text{in keys} \, (\text{NamesMap})$

RULE typeNamed-local-in-any-Phase

$\langle '\text{Name}(X:Id) \rangle_k \text{mapWrap} (X \mapsto \text{Class}:\text{ClassType} \rightarrow) \rightarrow_{\text{localTypes}}$

RULE typeNamed-global

$\langle '\text{Name}(X:Id) \rangle_k \text{mapWrap} (\text{LocalTypes}:\text{Map}) \rightarrow_{\text{localTypes}}$

$\langle \text{CrntClass}:\text{ClassType} \rangle_{\text{crntClass}} \langle \text{CrntClass} \rangle_{\text{classType}} \langle - X \mapsto \text{Class}:\text{ClassType} \rightarrow \rangle_{\text{imports}}$

REQUIRES $\neg \text{Bool} \, X \, \text{in keys} \, (\text{LocalTypes})$

RULE typeNamed-global-Fail

$\langle '\text{Name}(X:Id) \rangle_k \text{mapWrap} (\text{LocalTypes}:\text{Map}) \rightarrow_{\text{localTypes}}$

$\langle \text{CrntClass}:\text{ClassType} \rangle_{\text{crntClass}} \langle \text{CrntClass} \rangle_{\text{classType}} \langle \text{Imp}:\text{Map} \rangle_{\text{imports}}$

REQUIRES $\neg \text{Bool} \, X \, \text{in keys} \, (\text{LocalTypes}) \land \text{Bool} \, \neg \text{Bool} \, (X \, \text{in keys} \, (\text{Imp}))$

This two rules may only apply during processing of extends/implements clauses of top-level classes. When
the class whose declaration is processed is an inner class, usual rules for 'TypeName apply.

RULE typeNamed-in-ProcclassDecsPhase-Top

$\langle '\text{Name}(X:Id) \rangle_k \langle 'K \rangle_{\text{crntClass}} \langle - X \mapsto \text{Class}:\text{ClassType} \rightarrow \rangle_{\text{compUnitImports}}$

RULE typeNamed-in-ProcclassDecsPhase-Top-Fail

$\langle '\text{Name}(X:Id) \rangle_k \langle 'K \rangle_{\text{crntClass}} \langle \text{Imp}:\text{Map} \rangle_{\text{compUnitImports}}$

REQUIRES $\neg \text{Bool} \, X \, \text{in keys} \, (\text{Imp})$
C.12 Module ELABORATION-EXPRESSIONS

C.12.1 Numeric, boolean and String operators

Expression labels are not converted by the default rule in the phase `elabDispose() => elabRes(...)`. Each expression needs a specialized rule for disposing, that will compute, among others, the type of the expression.

**RULE ELABEND-BOOLRESULTBINARYEXP**

\[
\text{elabEnd} (KL:KLabel(TE1:K, TE2:K)) \\
\text{elabRes} ( \text{cast} (\text{bool}, KL(TE1, TE2)))
\]

**REQUIRES** `isBoolResultBinaryExpLabel(KL) = K true`

**SYNTAX**

\[
KItem ::= \text{isBoolResultBinaryExpLabel}(KLabel) \quad [\text{function}]
\]

**RULE**

\[
\text{isBoolResultBinaryExpLabel} (\text{'LazyOr}) \\
\text{true}
\]

**RULE**

\[
\text{isBoolResultBinaryExpLabel} (\text{'LazyAnd}) \\
\text{true}
\]

**RULE**

\[
\text{isBoolResultBinaryExpLabel} (\text{'Eq}) \\
\text{true}
\]

**RULE**

\[
\text{isBoolResultBinaryExpLabel} (\text{'NotEq}) \\
\text{true}
\]

**RULE**

\[
\text{isBoolResultBinaryExpLabel} (\text{'Lt}) \\
\text{true}
\]

**RULE**

\[
\text{isBoolResultBinaryExpLabel} (\text{'Gt}) \\
\text{true}
\]
RULE
isBoolResultBinaryExpLabel ("LtEq")
true

RULE
isBoolResultBinaryExpLabel ("GtEq")
true

RULE elabEnd-BoolOnlyResultExp-unary
elabEnd ("Not(TE:K)"

elabRes ( cast ( bool, "Not(TE)" ) )

RULE elabEnd-BinaryNumOperandNumResultExp
elabEnd (KL:KLabel(TE1:K, TE2:K))
elabExpAndType (KL(TE1, TE2), normalizeType ( type0f (TE1), type0f (TE2))

REQUIRES ( KL =KLabel 'Or ) \lor ( KL =KLabel 'ExcOr ) \lor ( KL =KLabel 'And ) \lor ( KL =KLabel 'Plus ) \lor ( KL =KLabel 'Minus ) \lor ( KL =KLabel 'Mul ) \lor ( KL =KLabel 'Div ) \lor ( KL =KLabel 'Remain )

RULE elabEnd-BitShift
elabEnd (KL:KLabel(TE1:K, TE2:K))
elabExpAndType (KL(TE1, TE2), normalizeType ( type0f (TE1))

REQUIRES ( KL =KLabel 'LeftShift ) \lor ( KL =KLabel 'RightShift ) \lor ( KL =KLabel 'URightShift )

RULE elabEnd-UnaryNumeric
elabEnd (KL:KLabel(TE1:K))
elabExpAndType (KL(TE1), normalizeType ( type0f (TE1)))

REQUIRES ( KL =KLabel 'Plus ) \lor ( KL =KLabel 'Minus ) \lor ( KL =KLabel 'Complement )

RULE
elabEnd ( ( ++ cast ( T1:Type, Exp1:K ) ) )
elabRes ( cast ( T1, ( ++ Exp1 ) ) )

RULE
elabEnd ( ( -- cast ( T1:Type, Exp1:K ) ) )
elabRes ( cast ( T1, ( -- Exp1 ) ) )
RULE

\[
\text{elabEnd} \left( \text{cast} \left( T1: \text{Type}, \text{Exp1}:K \right) \right) \\
\text{elabRes} \left( \text{cast} \left( T1, \text{Exp1} \right) \right)
\]

RULE

\[
\text{elabEnd} \left( \text{cast} \left( T1: \text{Type}, \text{Exp1}:K \right) \right) \\
\text{elabRes} \left( \text{cast} \left( T1, \text{Exp1} \right) \right)
\]

RULE \text{elabEnd-StringPlusAny}

\[
\text{elabEnd} \left( '\text{Plus}(\text{TE1}:K, \text{TE2}:K) \right) \\
\text{elabRes} \left( \text{cast} \left( \text{class String}, '\text{Plus}(\text{TE1}, \text{TE2}) \right) \right)
\]

REQUIRES\ typeOf \ (\text{TE1}) =_{K} \text{class String} \lor_{\text{Bool}} \text{typeOf} \ (\text{TE2}) =_{K} \text{class String}

Heats the second argument, that is reduced into a type. The whole expression is then rewritten into elabRes(cast(SecondArgAsType, FirstArg)) We cannot eliminate this wrapper because the second argument in all usage places is not a function.

SYNTAX \ KItem ::= \text{elabExpAndType} \ (K, K) \ [\text{strict}(2)]

RULE

\[
\text{elabExpAndType} \ (K:K, T:Type) \\
\text{elabRes} \left( \text{cast} \left( T, K \right) \right)
\]

C.12.2 Conditional operator

RULE \text{elabEnd-Cond}

\[
\text{elabEnd} \left( '\text{Cond}(\text{CondTE}:K, \text{TE1}:K, \text{TE2}:K) \right) \\
\text{elabExpAndType} \left( '\text{Cond}(\text{CondTE}, \text{TE1}, \text{TE2}), \text{condType} \ (\text{typeOf} \ (\text{TE1}), \text{typeOf} \ (\text{TE2})) \right)
\]

Computes the type of a conditional expression. Operands evaluate into types.

SYNTAX \ KItem ::= \text{condType} \ (\text{Type}, \text{Type}) \ [\text{strict}]

RULE \text{condType-NoChar}

\[
\text{condType} \ (\text{T1:Type}, \text{T2:Type}) \\
\text{ifAux} \ (\text{subtype} \ (\text{T1}, \text{T2}), \text{T2}, \text{T1})
\]

REQUIRES \ T1 \not_{K} \text{char} \land_{\text{Bool}} \ T2 \not_{K} \text{char}
RULE condType-SecondChar

condType ( T1:Type, char )

condType ( char, T1 )

REQUIRES T1 ≠ _K_ char

RULE condType-FirstChar

condType ( char, T2:Type )

ifAux ( subtype ( char, T2 ), T2, int )

C.12.3 Assignment operator

RULE elabEnd-Assign-SameType

elabEnd ('Assign( cast ( T1:Type, Exp1:T ), TExp2:T ) )

elabRes ( cast ( T1, 'Assign( Exp1, TExp2 ) ) )

REQUIRES T1 = _K_ type0f ( TExp2 )

RULE elabEnd-Assign-DiffType

elabEnd ( 'Assign( cast ( T1:Type, Exp1:T ), \( \frac{ TExp2:T }{ cast ( T1, TExp2 ) } \) ) )

REQUIRES T1 ≠ _K_ type0f ( TExp2 )

C.12.4 Cast operator

RULE

elabEnd ( cast ( T1:Type, Exp1:T ) *= TExp2:T )

elabRes ( cast ( T1, Exp1 *= TExp2 ) )

RULE

elabEnd ( cast ( T1:Type, Exp1:T ) /= TExp2:T )

elabRes ( cast ( T1, Exp1 /= TExp2 ) )

RULE

elabEnd ( cast ( T1:Type, Exp1:T ) %= TExp2:T )

elabRes ( cast ( T1, Exp1 %= TExp2 ) )
RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ +=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 += \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ -=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 -= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ <=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 <= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ >=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 >= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ &=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 &= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ ^=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 ^= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ ^=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 ^= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{cast}(T_1:Type,\ Exp_1:K) \ |=\ TE_2:K) \\
\quad \text{elabRes} \ (\text{cast}(T_1,\ \text{Exp}_1 |= \text{TE}_2))
\]

RULE
\[
\text{elabEnd} \ (\text{CastPrim}(T:Type,\ TExp:K)) \\
\quad \text{cast}(T,\ TExp)
\]

[structural]
C.12.5 Expressions over reference types

**RULE elabEnd-InstanceOf**

\[
\text{elabEnd} \left( \text{InstanceOf}(\text{TExp}:K, \text{RT2}:\text{RefType}) \right) \\
\text{elabRes} \left( \text{cast} \left( \text{true}, \text{InstanceOf}(\text{TExp}, \text{RT2}) \right) \right)
\]

Class literal types are heated by this rule.

**CONTEXT**

\[
\text{elab} \left( \text{Lit}('\text{Class}()) \right)
\]

**RULE elab-Lit-Class**

\[
\text{elab} \left( \text{Lit}('\text{Class}(\text{T}:\text{Type})) \right) \\
\text{elabRes} \left( \text{cast} \left( \text{class String2Id("java.lang.Class"), Lit('Class(T:Type))} \right) \right)
\]

C.13 Module ELABORATION-VARS

Elaboration of terms defined in var-lookup.k

Both unqualified and qualified AmbName.

**RULE elabEnd-AmbName**

\[
\text{elabEnd} \left( \text{AmbName(Ks):KList} \right) \\
\text{elabEnd} \left( \text{ExprName(Ks) ?? TypeName(Ks) ?? PackageName([ Ks ])} \right)
\]

C.13.1 Unqualified variable references

**RULE elabEnd-ExprName-localVar-ok**

\[
\text{elabEnd} \left( \text{ExprName(X:Id)} \right) \\
\text{elabRes} \left( \text{cast} \left( \text{T, ExprName(X)} \right) \right) \\
\text{mapWrap} \left( \text{X \mapsto T:Type} \right) \right)_{\text{elabEnv}}
\]
This could be either a field, or a local var of some enclosing block.

**SYNTAX**

\[KItem ::= \text{externalVar} \ (\ Id, \ ClassType)\]

**RULE externalVar**

\[\text{elabEnd} (\ 'Field (\ \text{cast} (\ \text{Class}, \ 'QThis (\ \text{Class})), \ X) \rangle \ \text{? ?} \ \text{elabOuterLocalVar} (\ X, \ \text{Class}) \ \text{? ?} \ \text{externalVar} (\ X, \ \text{EnclosingClass})\]

\[\langle \text{Class} \rangle_{\text{classType}} \langle \text{EnclosingClass} : \text{ClassType} \rangle_{\text{enclosingClass}}\]

**RULE externalVar-noClass**

\[\text{elabOuterLocalVar} (\ X : \ Id, \ \text{Class} : \text{ClassType})\]

\[\langle \text{CrntClass} \rangle_{\text{classType}} \langle \text{Class} \rangle_{\text{classType}} \langle \text{EnclosingLocalEnv} : \text{Map} \rangle_{\text{enclosingLocalEnv}}\]

**RULE elabOuterLocalVar-ok**

\[\langle \text{elabOuterLocalVar} (\ X : \ Id, \ \text{Class} : \text{ClassType}) \rangle_{\text{elabEnv}}\]

\[\langle \text{CrntClass} \rangle_{\text{classType}} \langle \text{Class} \rangle_{\text{classType}} \langle \text{EnclosingLocalEnv} : \text{Map} \rangle_{\text{enclosingLocalEnv}}\]

**RULE elabOuterLocalVar-not-found**

\[\langle \text{elabOuterLocalVar} (\ X : \ Id, \ \text{Class} : \text{ClassType}) \rangle_{\text{elabEnv}}\]

\[\langle \text{Class} \rangle_{\text{classType}}\]

\[\text{requires} \ \neg \text{Bool} \ (\ X \ \text{in} \ \text{keys} (\ \text{EnclosingLocalEnv}))\]
C.13.2 Self-references: this and A.this

RULE elabEnd-THIS
\[
\text{elabEnd} \left( \frac{'\text{This}('KList)}{'Q\text{This}(\text{Class})} \right) \xrightarrow{k} \text{Class:ClassType} \text{cntClass}
\]

RULE elabEnd-QTHIS-INSTANCECT
\[
\text{elabEnd} \left( '\text{QThis}('\text{Class:ClassType}) \right) \xrightarrow{k} \text{instance} \text{contextType}
\]

RULE elabEnd-QTHIS-STATICCT
\[
\text{elabEnd} \left( '\text{QThis}(-) \right) \xrightarrow{k} \text{static} \text{contextType}
\]

C.13.3 Fields

RULE elabEnd-ExprName-qualified
\[
\text{elabEnd} \left( 'Expr\text{Name}(\text{QualK,K}, X:Id) \right) \\
\text{elabEnd} \left( '\text{Field}(\text{QualK}, X) \right)
\]

RULE elabEnd-Field-of-package
\[
\text{elabEnd} \left( '\text{Field}(---:\text{PackageId}, ---:Id) \right) \\
\xrightarrow{'K} \text{[structural]}
\]

RULE elabEnd-Field-of-noValue
\[
\text{elabEnd} \left( '\text{Field}('K, ---:Id) \right) \\
\xrightarrow{'K} \text{[structural]}
\]

RULE elabEnd-Field-QualRef
\[
\text{elabEnd} \left( '\text{Field}(\text{cast} \left( \text{Class:ClassType, Qual:K} \right), X:Id) \right) \\
\text{elabFieldImpl} \left( \text{cast} \left( \text{Class, Qual} \right), \text{elabLookup} \left( X, \text{Class} \right) \right)
\]

RULE elabEnd-Field-QualClass
\[
\text{elabEnd} \left( '\text{Field}(\text{Class:ClassType, X:Id}) \right) \\
\text{elabFieldImpl} \left( 'K, \text{elabLookup} \left( X, \text{Class} \right) \right)
\]
Searches the given field name in the given type (set of types), both static and instance context.

Important!!! The name is searched in this type and base types only. It is not searched in outer classes.
That is the job of externalVar().

**Syntax**

\[ \text{KItem ::= elabLookup ( Id, ClassType )} \]
\[ \text{| elabLookup ( Id, Set )} \]

**Rule elabLookup-Found-instance**

\[
\text{elabLookup ( X:Id, Class:ClassType )} \rightarrow k \langle \text{Class} \rangle_{\text{classList}}
\]
\[
\text{fieldEntry ( Class, X, T, instance )}
\]
\[
\text{[ [ \_ _, \text{FieldDec}([ \_ , T:Type, \text{VarDec(X) } ]), \_ ] \_ ]}_{\text{instanceFields}}
\]

**Rule elabLookup-Found-static**

\[
\text{elabLookup ( X:Id, Class:ClassType )} \rightarrow k \langle \text{Class} \rangle_{\text{classList}}
\]
\[
\text{fieldEntry ( Class, X, T, static )}
\]
\[
\text{[ [ \_ _, \text{FieldDec}([ \_ , \text{Static}(), T:Type, \text{VarDec(X) } ]), \_ ] \_ ]}_{\text{staticFields}}
\]

**Rule elabLookup-Found-constant**

\[
\text{elabLookup ( X:Id, Class:ClassType )} \rightarrow k \langle \text{Class} \rangle_{\text{classList}}
\]
\[
\text{TV}
\]
\[
\text{[ [ \_ X \rightarrow TV:TypedVal \_ ] \_ ]}_{\text{constantEnv}}
\]

If X is not found in the current class, search for it first in base interfaces, then in the base class. This order is necessary to avoid the case when base class has a private field X, and base interfaces have a public one. In this case we should choose the field from the interface.

**Rule elabLookup-NextClass**

\[
\text{elabLookup ( X:Id, Class:ClassType )} \rightarrow k \langle \text{Class} \rangle_{\text{classList}}
\]
\[
\text{elabLookup ( X, BaseInterfaces ) ?? elabLookup ( X, BaseClass )}
\]
\[
\text{[ [ InstanceVarDecs:KList ] \_ ]}_{\text{instanceFields}} \text{[ [ StaticVarDecs:KList ] \_ ]}_{\text{staticFields}}
\]
\[
\text{[ [ ConstantEnv:Map ] \_ ]}_{\text{constantEnv}}
\]

\[ \text{REQUIRES} \neg_{\text{Bool}} ( \text{nameInVarDecs ( X, [ InstanceVarDecs ] ) =}_{K}\text{true} \lor_{\text{Bool}} \text{nameInVarDecs ( X, [ StaticVarDecs ] ) =}_{K}\text{true} \lor_{\text{Bool}} ( X \text{ in keys ( ConstantEnv )} )) \]

**Rule elabLookup-NotFound**

\[
\text{elabLookup ( X:Id, \_ )}
\]
\[ \_K \]
RULE elabLookup-Set

elabLookup (X:Id, Class:ClassType  Rest:Set)

elabLookup (X, Class) ?? elabLookup (X, Rest)

RULE elabLookup-Set-NotFound

elabLookup (—, 'Set)

rule elabFieldImpl

elabFieldImpl (cast (Class, 'QThis(Class)), elabLookup (X, BaseClass))

for constant fields
\[
\text{rule}
\]
\[
\text{elabFieldImpl} \quad (-, \text{TV} : \text{TypedVal})
\]
\[
\quad \text{TV}
\]

\[
\text{rule}
\]
\[
\text{elabFieldImpl} \quad (-, \`\kappa)
\]
\[
\quad \`\kappa
\]

C.14 Module ELABORATION-METHOD-INVOC

C.14.1 Regular method calls

\[
\text{rule}
\]
\[
\text{customElabEnd} \quad (\text{'Method})
\]
\[
\quad \text{true}
\]

\[
\text{rule}
\]
\[
\text{elabEnd} \quad (\text{'Method} (\text{Qual} : \text{K}, \text{Name} : \text{Id})
\]
\[
\quad \text{elabEnd} \quad (\text{'Method} (\text{MethodName} (\text{Qual}, \text{Name})))
\]
\[
\quad \text{[structural]}
\]

\[
\text{rule elabEnd-Method-MethodName-Unq-InstanceCT}
\]
\[
\quad \text{elab} \quad (\text{'Method} (\text{MethodName} (\text{Name} : \text{Id}))
\]
\[
\quad \text{elabEnd} \quad (\text{'Method} (\text{MethodName} (\text{Name} : \text{Id})))
\]
\[
\quad (\text{Class} : \text{ClassType} \text{crntClass}
\]
\[
\quad \text{[structural]}
\]

\[
\text{rule elabEnd-Method-MethodName-to-elabRes}
\]
\[
\quad \text{elabEnd} \quad (\text{'Method} (\text{MethodName} (-: \text{K}, \text{Name} : \text{Id})))
\]
\[
\quad \text{'elabRes}
\]

Computes into an expression of the form elabRes(cast(QualClass, 'QThis(QualClass))), where QualClass is searched in the enclosing context of the first argument, being the innermost class that has a method with name = the second argument. Or noValue if no suitable result is found.

This algorithm is correct even when both inner and outer class have methods with the same name. Java don’t support method overloading in the context of nesting. In other words, inner class methods don’t overload but rather hide methods with the same name from outer classes. See program inner_in_81_inner_overl_outer.java as example.

287
SYNTAX \( KItem ::= \text{findQualifierForName} \left( \text{ClassType}, \text{Id} \right) \)

RULE

\[
\text{isElabNaked} \left( \text{findQualifierForName} \left( \_, \_ \right) \right) \quad \text{true}
\]

RULE elabEnd-findQualifierForName

\[
\text{findQualifierForName} \left( \text{QualClass:ClassType}, \text{Name:Id} \right)
\begin{array}{c}
\text{ifAux} \left( \text{containsName} \left( \text{Methods, Name} \right) =_k \text{true}, \text{elabRes} \left( \text{cast} \left( \text{QualClass,} \\
\text{'QThis(QualClass)})}, \text{findQualifierForName} \left( \text{QualEncloserClass, Name} \right) \right) \right)
\end{array}
\]

\[
\begin{array}{c}
\text{enclosingClass} \left( \text{Methods:Map} \right) \\
\text{classType} \left( \text{QualClass} \right)
\end{array}
\]

SYNTAX \( KItem ::= \text{containsName} \left( \text{Map}, \text{Id} \right) \) [function]

RULE

\[
\text{containsName} \left( \left( \text{sig} \left( \text{Name, _} \rightarrow _\_ \right) \\
\rightarrow \text{Map, Name:Id} \right) \right) \quad \text{true}
\]

Happens for 'NewInstance expressions for static inner classes.

RULE elabEnd-findQualifierForName-static

\[
\text{findQualifierForName} \left( \_K, K.K \right) \\
\]

C.14.2 Superclass method calls: super.m()

RULE

\[
\text{customElabEnd} \left( \SimSuperMethod \right) \\
\quad \text{true}
\]

\[
\text{super.X(\_)} \\
\text{super.X(\_)}
\]

RULE SuperMethod-desugar

\[
\begin{array}{c}
\text{elabEnd} \left( \SimSuperMethod(K.K, \text{Name:Id}) \right) \\
\text{elabEnd} \left( \SimQSuperMethod(Class, K.K, \text{Name:Id}) \right)
\end{array}
\]

\[
\begin{array}{c}
\left( \text{Class:ClassType} \right) \text{crntClass}
\end{array}
\]

[structural]
rule elabEnd Invoke QSuperMethod to lookupMethod

\[
\kappa \quad \overset{\kappa}{\text{elabEnd}} \left('\text{Invoke}^-_k\right)
\]

lookupMethod (BaseClass, true, Name, [ ArgExps ])

\[
'Q\text{SuperMethod}(\text{Class}:(\text{ClassType}, -: :K, \text{Name}:Id), [ \text{ArgExps}:\text{KList} ]))
\]

\( \text{Class} \langle \text{classType} \langle \text{BaseClass}:\text{ClassType} \rangle \rangle \text{extends} \)

rule methodLookupResult Invoke QSuperMethod static end

\[
\overset{\kappa}{\text{elabEnd}} \left('\text{Invoke}(\text{QSuperMethod}(\text{Class}:(\text{ClassType}, -: :K, \text{Name}:Id)), [ \text{ArgExps} ])\right)
\]

\( \text{methodLookupResult} (\text{RetT}::\text{Type}, [ \text{ExpectedArgExps}:\text{KList} ], \text{static}) \overset{\kappa}{\text{elabEnd}} ('\text{Invoke}(\text{QSuperMethod}(\text{Class}:(\text{ClassType}, -: :K, \text{Name}:Id)), [ \text{ExpectedArgExps} ]))\)

\( \langle \text{Class} \langle \text{classType} \langle \text{BaseClass}:\text{ClassType} \rangle \rangle \text{extends} \)

\rule methodLookupResult Invoke QSuperMethod instance end

\[
\overset{\kappa}{\text{elabEnd}} \left('\text{Invoke}(\text{QSuperMethod}(\text{Class}:(\text{ClassType}, :K_2:K, \text{Name}:Id)), [ \text{ExpectedArgExps} ])\right)
\]

\( \text{methodLookupResult} (\text{RetT}::\text{Type}, [ \text{ExpectedArgExps}:\text{KList} ], \text{instance}) \overset{\kappa}{\text{elabEnd}} ('\text{Invoke}(\text{QSuperMethod}(\text{Class}:(\text{ClassType}, :K_2:K, \text{Name}:Id)), [ \text{ExpectedArgExps} ]))\)

\( \langle \text{Class} \langle \text{classType} \langle \text{BaseClass}:\text{ClassType} \rangle \rangle \text{extends} \)

rule elabEnd Invoke to lookupMethod

\[
\kappa
\]

lookupMethod (getLookupTargetType (typeOf Qual), isCastOfQThis (Qual) = \kappa true, Name, [ ArgExps ])

\( \overset{\kappa}{\text{elabEnd}} ('\text{Invoke}(\text{Method}(\text{MethodName}(\text{Qual}:K, \text{Name}:Id)), [ \text{ArgExps}:\text{KList} ]))\)

\( \text{Returnts the same type for classes, arrayImplClass for arrays.} \)

SYNTAX \( \text{KItem ::= getLookupTargetType (RefType) \ [function]} \)

RULE

getLookupTargetType (Class:ClassType)

\( \text{Class} \)

RULE

getLookupTargetType (arrayOf --)

\( \text{classArrayImpl} \)

SYNTAX \( \text{KItem ::= isCastOfQThis (K) \ [function]} \)

289
C.14.3 Method reference lookup

Lookup the right methodRef() based on supplied parameters. If the method is not found in the supplied class list, we should search for it in the class list corresponding to the enclosing class of the caller class. First we search for the signature — lookupSignature construct. Afterwards we cast the actual arguments to the types expected by the found method signature.

**Syntax**

\[\text{KItem ::= lookupMethod} ( \text{ClassType, Bool, Id, KListWrap} )\]

\[\text{KItem ::= lookupMethod} ( \text{ClassType, Bool, Id, KListWrap, MethodRef, K} ) \quad \text{[strict(5, 6)]}\]

**Rule**

\[\text{lookupMethod} ( \text{QualClass:ClassType, QualIsThis:Bool, MethodName:Id, [ ArgExps:KList ]} )\]

\[\text{lookupMethod} ( \text{QualClass:ClassType, QualIsThis:Bool, MethodName:Id, [ ArgExps }, \cdot, \text{K, tempKResult } )\]

A Temporary KResult that prevents this term to be heated before it is transformed by another rule into a proper term to be heated. This is not a final KResult in its surrounding context. Basically, this term is used to cheat K heating rules.

**Syntax**

\[\text{KResult ::= tempKResult}\]

**Rule lookupMethod-to-lookupSignature**

\[\text{lookupMethod} ( \text{QualClass:ClassType, QualIsThis:Bool, MethodName:Id, [ ArgExps:KList ]}, \cdot, \text{K}\)

\[\text{lookupSignature} ( \text{MethodName, getTypes} ( [ \text{ArgExps} ] ), \text{Methods, } \text{QualIsThis, } \cdot, \text{K, QualClass} ), \text{tempKResult } )\]

\[\langle \text{QualClass } \text{classType } \langle \text{Methods:Map } \text{methods} \rangle \]

**Rule lookupMethod-SigFound**

\[\text{lookupMethod} ( \cdot, \cdot, \cdot, [ \text{ArgExps:KList } ], \text{methodRef} ( \text{sig} ( \text{Name:Id, ArgTypes:Types} )\]

\[\cdot, \cdot ) , \text{tempKResult }\]

\[\text{methodCastArguments} ( \text{ArgTypes, [ ArgExps }, \cdot, , \text{'}KList list } )\]

**Syntax**

\[\text{KItem ::= methodCastArguments} ( \text{Types, KListWrap, KListWrap} )\]
RULE METHODCASTARGUMENTS-PROCESS

\[
\text{methodCastArguments} \left( \text{types} \left( \begin{array}{c}
\text{ArgType}\colon\text{Type} \\
\text{KList}
\end{array} \right), \quad \text{ParamExp}\colon K \\
\text{KList}
\right)
\]

\[
\text{cast} \left( \text{ArgType}, \text{ParamExp} \right)
\]

RULE METHODCASTARGUMENTS-END

\[
\text{methodCastArguments} \left( \text{types} \left( \begin{array}{c}
\text{KList} \\
\text{KList}
\end{array} \right), \quad \text{ExpectedArgExps}\colon\text{KList} \right)
\]

\[
\text{kr} \left[ \text{ExpectedArgExps} \right]
\]

RULE LOOKUPEMETHOD-END

\[
\text{lookupMethod} \left( \text{MethodRef} \left( \begin{array}{c}
\text{Sig}\colon\text{Signature} \\
\text{Class}\colon\text{ClassType}
\end{array} \right)
\right)
\]

\[
\text{methodLookupResult} \left( \text{RetT}, \text{ExpectedArgExps}, \text{CT} \right)
\]

\[
\text{methodSignature} \quad \text{methodReturnType} \\
\text{methodContextType}
\]

Marker indicating that arguments of the following 'Invoke, 'QNewInstance, 'QSuperConstrInv, 'AltConstrInv have been elaborated.

SYNTAX \( KItem ::= \text{methodLookupResult} \ (\text{Type}, \text{KListWrap}, \text{ContextType}) \)

In the past implementation, the non-empty qualifier of a static method call was moved to a separate statement in front of a method call using stmtAndExp(). But since stmtAndExp() is incompatible with Java intermediate form, we keep the non-empty qualifier as is even for static methods.

RULE METHODPROCESSARGUMENTS-INVOLVE-END

\[
\text{methodLookupResult} \left( \text{RetT}, \text{ExpectedArgExps}, \text{KList}, \text{CT} \right)
\]

\[
\text{elabEnd} \left( \text{cast} \left( \text{RetT}, \text{MethodRef} \left( \begin{array}{c}
\text{MethodName}\colon\text{Id} \\
\text{Name}\colon\text{Id}
\end{array} \right)
\right), \quad \text{if} \ \text{CT} = \text{K static}
\quad \text{isCastOfQThis} \ (\text{QualK}) = \text{K true} \quad \text{then} \quad \text{typeOf} \ (\text{QualK}) \quad \text{else} \quad \text{QualK} \ #fi, \text{Name}\colon\text{Id}
\right)
\]

C.14.4 Method signature lookup

SYNTAX \( KItem ::= \text{lookupSignature} \ (\text{Id}, \text{K}, \text{Map}, \text{Bool}, \text{MethodRef}, \text{ClassType}) \) [strict(2,5)]
**RULE lookupSignature-Main**

\[
\text{lookupSignature} (\text{MethodName} : \text{Id}, \text{CallTs} : \text{Types}, \\
\quad \text{sig} (\text{MethodName}, \text{SigTs} : \text{Types}) \mapsto \text{NewDecClass} : \text{ClassType} \\
\quad \mapsto : \text{Map}, \, \text{IsQ} : \text{Bool}, \\
\quad \text{OldMethodRef} : \text{MethodRef} \\
\quad \text{ifAux} (\text{andAux} (\text{andAux} (\text{subtypeList} (\text{CallTs}, \text{SigTs}), \#\text{if} \text{OldMethodRef} \\
\quad = \text{K} \, \#\text{then true} \#\text{else subtypeList} (\text{SigTs}, \text{getMethodRefArgTypes} (\text{OldMethodRef}))) \#\text{fi} ), \text{isAccessible} (\text{NewAccMode}, \text{NewCT}, \\
\quad \text{NewDecClass}, \text{IsQ}, \text{QualClass})), \text{methodRef} (\text{sig} (\text{MethodName}, \text{SigTs}), \\
\quad \text{NewDecClass}), \text{OldMethodRef} \\
\quad \text{, QualClass} : \text{ClassType} \rangle \\
\langle \text{NewDecClass} : \text{classType} \langle \text{sig} (\text{MethodName}, \text{SigTs}) \rangle \text{methodSignature} \\
\langle \text{NewAccMode} : \text{AccessMode} \rangle \text{methodAccessMode} \langle \text{NewCT} : \text{ContextType} \rangle \text{methodContextType} \\
\text{SYNTAX} \quad \text{KItem} ::= \text{getMethodRefArgTypes} (\text{MethodRef}) \ [\text{function}] \\
\text{RULE} \\
\text{getMethodRefArgTypes} (\text{methodRef} (\text{sig} (\text{---}, \text{Ts} : \text{Types}), \text{---})) \\
\downarrow \text{Ts} \\
\text{RULE lookupSignature-SigDiscard} \\
\text{lookupSignature} (\text{MethodName} : \text{Id}, \text{---}, \\
\quad \text{sig} (\text{Name} : \text{Id}, \text{---}) \mapsto \text{---} : \text{Map}, \text{---} : \text{Map} \\
\quad \text{REQUIRES} \text{Name} \neq \text{K} \text{MethodName} \\
\text{RULE lookupSignature-End} \\
\text{lookupSignature} (\text{---} : \text{Map}, \text{---} : \text{Map}, \text{MethodRef} : \text{MethodRef}, \text{---} : \text{Map}) \\
\downarrow \text{MethodRef} \\
\text{C.14.5 \ Method accessibility check} \\
\text{Interaction between overloading and access modes.} \\
\text{Tests whether a method from a given class with a given AccessMode and ContextType may be called from the current object environment.} \\
\text{SYNTAX} \quad \text{KItem} ::= \text{isAccessible} (\text{AccessMode}, \text{K}, \text{ClassType}, \text{Bool}, \text{ClassType}) \ [\text{strict}(1, 2)]
RULE `isAccessible-public`

\[
\text{isAccessible ( public, —, —, —, —)}
\]

true

RULE `isAccessible-protected`

\[
\text{isAccessible ( protected, CT:ContextType, DeclaringClass:ClassType, IsQ:Bool, TargetClass:ClassType)}
\]

\[
\begin{align*}
\text{orAux ( andAux ( orAux ( CT =K \text{ static, IsQ =K false :: bool }, subtype ( ifAux ( IsQ, CurrentClass, TargetClass, DeclaringClass)), isAccessible ( package, CT, DeclaringClass, IsQ, TargetClass))}
\end{align*}
\]

\[
\begin{align*}
\text{CurrentClass:ClassType} \langle\langle\langle \text{CurrentClass} \rangle\rangle\rangle_{\text{crntClass}}
\end{align*}
\]

RULE `isAccessible-package`

\[
\text{isAccessible ( package, —, DeclaringClass:ClassType, —, —)}
\]

\[
\begin{align*}
eqAux ( \text{getPackage ( getTopLevel ( DeclaringClass)), getPackage ( getTopLevel ( CurrentClass))}
\end{align*}
\]

\[
\begin{align*}
\text{CurrentClass:ClassType} \langle\langle\langle \text{CurrentClass} \rangle\rangle\rangle_{\text{crntClass}}
\end{align*}
\]

RULE `isAccessible-private`

\[
\text{isAccessible ( private, —, DeclaringClass:ClassType, —, —)}
\]

\[
\begin{align*}
eqAux ( \text{getTopLevel ( DeclaringClass), getTopLevel ( CurrentClass))}
\end{align*}
\]

\[
\begin{align*}
\text{CurrentClass:ClassType} \langle\langle\langle \text{CurrentClass} \rangle\rangle\rangle_{\text{crntClass}}
\end{align*}
\]

C.15 Module ELABORATION-NEW-INSTANCE

elab(‘NewInstance()) is strict in arguments 2 and 3 — class name and constructor arguments. Constructor arguments have to be heated if this class is anonymous.

RULE

\[
\text{customElabHeating (‘NewInstance, [ Ks:KList ] )}
\]

\[
\begin{align*}
\text{length ( [ Ks ] ) in ( 2 3)}
\end{align*}
\]

RULE `elab-NewInstance`

\[
\text{elab (}
\]

\[
\begin{align*}
\text{‘NewInstance(Arg1:K, Class:ClassType, ActualArgsList:K, ‘None(KList))}
\end{align*}
\]

\[
\begin{align*}
\text{‘QNewInstance(#if CT =K \text{ static } \#then ‘K \#else findQualifierOfType ( CrntClass, EnclosingClass) #fi, Arg1, Class, ‘None(KList), ActualArgsList, ‘None(KList))}
\end{align*}
\]

\[
\begin{align*}
\text{CrntClass:ClassType} \langle\langle\langle \text{CrntClass} \rangle\rangle\rangle_{\text{crntClass}} \langle\langle\langle \text{Class} \rangle\rangle\rangle_{\text{classType}} \langle\langle\langle \text{EnclosingClass:ClassType} \rangle\rangle\rangle_{\text{enclosingClass}} \langle\langle\langle \text{CT:ContextType} \rangle\rangle\rangle_{\text{classContextType}}
\end{align*}
\]

293
elab('QNewInstance()) is strict in all its arguments except the class name (3) and anonymous body(6):

- Argument (3) is heated if it is not a simple name but a precursor to a type
- Argument (6) is heated if it is `None() — an empty body.

**Rule customElabHeating**

\[
\text{length (\left[\text{Ks, KHole}\right]\right))} \in (1 2 4 5) \\
\neg_{\text{Bool}} \exists_{\text{Id}}(\text{KHole}) =_{\text{K}} \text{true} \\
\text{length (\left[\text{Ks, KHole}\right]\right))} =_{\text{K}} 6 \\
\neg_{\text{Bool}} \text{KHole} =_{\text{K}} 'None(\text{'}KList\text{'})
\]

**Rule elab-QNewInstance-resolve-class**

\[
\text{elab ('QNewInstance(elabRes (\text{cast (QualClass:ClassType, -)}), -):K,}
\]

\[
\text{resolveInnerClass (QualClass, Name)}
\]

Resolve the simple name of an inner class to a fully qualified class name in a qualified new like: o.new A(...);

**Syntax**

\[
KItem ::= \text{resolveInnerClass (ClassType, Id)}
\]

**Rule isElabNaked**

\[
\text{resolveInnerClass (-, -)}
\]

\[
\text{true}
\]

**Rule resolveInnerClass**

\[
\text{resolveInnerClass (QualClass:ClassType, Name:Id)}
\]

\[
\text{getClassIfExists (toPackage (QualClass, Name)) ?? resolveInnerClass (}-
\]

\[
\text{BaseQualClass, Name)}
\]

\[
\langle \text{QualClass}_{\text{classType}} \langle \text{BaseQualClass:ClassType}\rangle_{\text{extends}}
\]

If with the given package and name exists, return this class. Otherwise return noValue

**Syntax**

\[
KItem ::= \text{getClassIfExists (PackageId, Id)}
\]

**Rule getClassIfExists**

\[
\text{getClassIfExists (Pack:PackageId, Name:Id)}
\]

\[
\#\text{if Name in keys (NamesToClasses) } #\text{then getClassType (Pack:PackageId, Name:Id) } #\text{else } #\text{fi}
\]

\[
\langle - \text{Pack:PackageId} \map\text{\_wrap (NamesToClasses:Map) } -\rangle_{\text{namesToClasses}}
\]
QNewInstance lookup is always qualified (e.g. not this), because the object we are creating is different from the current object.

All new instance creation expressions are converted into qualified ones — 'QNewInstance, during elaboration phase. For instance inner classes, the qualifier will be a valid expression for the qualifier. For other classes qualifier will be noValue. At this stage 'QNewInstance is wrapped in elabEnd.

A 'QNewInstance with target noValue should be packed back into a NewInstance, for compatibility with Java. It will be desugared again into 'NewInstance at execution phase.

C.16 Module ELABORATION-ARRAYS

C.16.1 Desugaring of c-style array declarators

C-style array declaration Applied in both elaboration and execution phase

```
RULE elab-LocalVarDec-ArrayVarDecId-desugar
  elab ('LocalVarDec(—:K, T:Type, , —:KList ) ) )
  elabRes ( 'VarDec('ArrayVarDecId(X:Id, [ Dim(KList )], — ) )
    arrayOf T
  ) )
```

rule elab-LocalVarDec-ArrayVarDecId-discard

\[
\text{elab} \left( \text{LocalVarDec}(---:K, \ T:\text{Type}, \ [ \text{VarDec}(---:K, \ T:\text{Type}, \ [ \text{ArrayVarDecId}(X:\text{Id}, \ [\text{KList}] \ ), \ --:KList] \ ))) \right)
\]

[structural]

rule elab-LocalVarDec-ArrayInit-desugar

\[
\text{elab} \left( \text{LocalVarDec}(---:K, \ T:\text{Type}, \ [ \text{VarDec}(X:\text{Id}, \ [ \text{ArrayInit}(\text{InitK}:K) \ ), \ 'NewArray(T, \ [\text{KList}], \ 'ArrayInit(\text{InitK})] \ )) \right)
\]

[structural]

rule FieldDec-ArrayVarDecId-desugar

\[
\text{FieldDec}(---:K, \ T:\text{Type}, \ [ \text{VarDec}(\text{ArrayVarDecId}(X:\text{Id}, \ [\text{KList}] \ ), \ 'Dim(\text{KList}) \ ), \ --:KList] \ ]) \}
\]

[structural]

rule FieldDec-ArrayVarDecId-discard

\[
\text{FieldDec}(---:K, \ T:\text{Type}, \ [ \text{VarDec}(\text{ArrayVarDecId}(X:\text{Id}, \ [\text{KList}] \ ), \ --:KList] \ ]) \}
\]

[structural]

rule FieldDec-ArrayInit-desugar

\[
\text{FieldDec}(---:K, \ T:\text{Type}, \ [ \text{VarDec}(\text{ArrayInit}(\text{InitK}:K) \ ), \ 'NewArray(T, \ [\text{KList}], \ 'ArrayInit(\text{InitK})] \ )) \}
\]

[structural]

rule Param-ArrayVarDecId-desugar

\[
\text{Param}(---:K, \ T:\text{Type}, \ [ \text{ArrayVarDecId}(X:\text{Id}, \ [\text{KList}] \ )) \]
\]

[structural, anywhere]

rule Param-ArrayVarDecId-Discard

\[
\text{Param}(---:K, \ T:\text{Type}, \ [ \text{ArrayVarDecId}(X:\text{Id}, \ [\text{KList}] \ ] \ ) \]
\]

[structural, anywhere]

Thi rule and the next one may happen both in elaboration, execution or process members phases. They are required to be [anywhere].

rule Param-ArrayVarDecId-desugar

\[
\text{Param}(---:K, \ T:\text{Type}, \ [ \text{ArrayVarDecId}(X:\text{Id}, \ [\text{KList}] \ )) \]
\]

[structural, anywhere]
C.16.2 Main array-related expressions

RULE elabEnd-ArrayAccess
   elabEnd ('ArrayAccess(cast ('array0f T:Type, TargetExp:K), IndexTE:K))
   \elabRes ( \cast (T, 'ArrayAccess(cast ('array0f T, TargetExp), IndexTE)))

RULE elabEnd-Field-ArrayLength
   elabEnd ('Field(cast ('array0f T:Type, Qual:K), X:Id))
   \elabRes ( \cast (int, 'Field(cast ('array0f T, Qual), X)))
REQUIRES Id2String (X) == String "length"

RULE elabEnd-NewArray-EmptyDims
   elabEnd ('NewArray(T:Type, [ Dim (KList)]), [ Dim (KList)]))
   [structural]

RULE elabEnd-NewArray
   elabEnd ('NewArray(T:Type, [ Dims:KList], [ KList ]))
   \elabExpAndType ('NewArray(T, [ Dims ], [ KList ]), \getArrayType (T, [ Dims ]))
   [structural]

Computes the array type based on allocated elem type and number of allocated dimensions.

SYNTAX  KItem ::= \getArrayType (Type, KListWrap) [function]

RULE
   \getArrayType (T:Type, [ K:K, Dims:KList ])
   \getArrayType (array0f T, [ Dims ])

RULE
   \getArrayType (T:Type, [ KList ])
   T

RULE elabEnd-NewArray-ArrayInit-Preprocess
   elabEnd ('NewArray(T:Type, [ Dim (KList)]), [ Dim (KList)]), ArrayInit(-)))

297
**RULE elabEnd-NewArray-ArrayInit**

```plaintext
elabEnd ('NewArray(arrayOf T:Type, [ 'KList ], 'ArrayInit([ InitContent:KList ])))
```

evalRes ( cast ( arrayOf T, 'NewArray(arrayOf T:Type, [ 'KList ], 'ArrayInit( [ InitContent:KList ]))) )

## C.17 Module ELABORATION-CATEGORIES

This module contains a few auxiliary functions governing what AST labels should be processed by default rules for elaboration, and which ones need custom rules.

Java KLabels that are processed by default heating/cooling rules of elaboration. All KLabels that can be part of a code block during elaboration phase, except those members of customElabChildren or isElabNaked groups.

This predicate should be disjoint with customElabChildren (no longer used) and isElabNaked

**SYNTAX**

```plaintext
KItem ::= defaultElabHeating ( KLabel ) [function]
```

**RULE**

```plaintext
defaultElabHeating ('Some)
true
```

**RULE**

```plaintext
defaultElabHeating ('None)
true
```

**RULE**

```plaintext
defaultElabHeating ('Assign)
true
```

**RULE**

```plaintext
defaultElabHeating ('AssignMul)
true
```

**RULE**

```plaintext
defaultElabHeating ('AssignDiv)
true
```

**RULE**

```plaintext
defaultElabHeating ('AssignRemain)
true
```
RULE
defaultElabHeating ('AssignPlus)
true

RULE
defaultElabHeating ('AssignMinus)
true

RULE
defaultElabHeating ('AssignLeftShift)
true

RULE
defaultElabHeating ('AssignRightShift)
true

RULE
defaultElabHeating ('AssignURightShift)
true

RULE
defaultElabHeating ('AssignAnd)
true

RULE
defaultElabHeating ('AssignExcOr)
true

RULE
defaultElabHeating ('AssignOr)
true

RULE
defaultElabHeating ('InstanceOf)
true
RULE

defaultElabHeating ('Mul')
    true

RULE

defaultElabHeating ('Div')
    true

RULE

defaultElabHeating ('Remain')
    true

RULE

defaultElabHeating ('Plus')
    true

RULE

defaultElabHeating ('Minus')
    true

RULE

defaultElabHeating ('LeftShift')
    true

RULE

defaultElabHeating ('RightShift')
    true

RULE

defaultElabHeating ('URightShift')
    true

RULE

defaultElabHeating ('Lt')
    true
RULE
defaultElabHeating ('Gt')
   ---------------
      true

RULE
defaultElabHeating ('LtEq')
   ---------------
      true

RULE
defaultElabHeating ('GtEq')
   ---------------
      true

RULE
defaultElabHeating ('Eq')
   ---------------
      true

RULE
defaultElabHeating ('NotEq')
   ---------------
      true

RULE
defaultElabHeating ('LazyAnd')
   ---------------
      true

RULE
defaultElabHeating ('LazyOr')
   ---------------
      true

RULE
defaultElabHeating ('And')
   ---------------
      true

RULE
defaultElabHeating ('ExcOr')
   ---------------
      true
RULE
  defaultElabHeating ('Or')
  true

RULE
  defaultElabHeating ('Cond')
  true

RULE
  defaultElabHeating ('PreIncr')
  true

RULE
  defaultElabHeating ('PreDecr')
  true

RULE
  defaultElabHeating ('Complement')
  true

RULE
  defaultElabHeating ('Not')
  true

RULE
  defaultElabHeating ('CastPrim')
  true

RULE
  defaultElabHeating ('CastRef')
  true

RULE
  defaultElabHeating ('PostIncr')
  true
RULE

defaultElabHeating ("PostDecr")

true

RULE

defaultElabHeating ("Invoke")

true

RULE

defaultElabHeating ("Method")

true

RULE

defaultElabHeating ("SuperMethod")

true

RULE

defaultElabHeating ("QSuperMethod")

true

RULE

defaultElabHeating ("GenericMethod")

true

RULE

defaultElabHeating ("ArrayAccess")

true

RULE

defaultElabHeating ("Field")

true

RULE

defaultElabHeating ("SuperField")

true
RULE
defaultElabHeating ('QSuperField)

true

RULE
defaultElabHeating ('NewArray)

true

RULE
defaultElabHeating ('Dim)

true

RULE
defaultElabHeating ('This)

true

RULE
defaultElabHeating ('QThis)

true

RULE
defaultElabHeating ('AltConstrInv)

true

RULE
defaultElabHeating ('QSuperConstrInv)

true

RULE
defaultElabHeating ('Empty)

true

RULE
defaultElabHeating ('Labeled)

true
RULE
defaultElabHeating ('ExprStm)

RULE
defaultElabHeating ('If)

RULE
defaultElabHeating ('AssertStm)

RULE
defaultElabHeating ('Switch)

RULE
defaultElabHeating ('SwitchBlock)

RULE
defaultElabHeating ('SwitchGroup)

RULE
defaultElabHeating ('Case)

RULE
defaultElabHeating ('Default)

RULE
defaultElabHeating ('While)
RULE

defaultElabHeating ('DoWhile)
true

RULE

defaultElabHeating ('Break)
true

RULE

defaultElabHeating ('Continue)
true

RULE

defaultElabHeating ('Return)
true

RULE

defaultElabHeating ('Throw)
true

RULE

defaultElabHeating ('Synchronized)
true

RULE

defaultElabHeating ('Try)
true

RULE

defaultElabHeating ('NoMethodBody)
true

RULE

defaultElabHeating ('ArrayInit)
true
Naked terms are those that should be computed directly into KResult during elaboration. Those are literals, types and packages. They are heated 'as is', without being wrapped into `elab()`. An exception is the class literal that is not executed during elaboration.

Some auxiliary functions are also included in this category in order to reduce the usage of elaboration wrappers.

**SYNTAX**

\[ KItem ::= \text{isElabNaked} (K) \] [function]
Terms that should use custom elaboration rules. For those terms:

- They will not be automatically heated from their parents into the \texttt{elab()} state.
- They will not be automatically passed to \texttt{elabDispose()} state. Instead, those terms should have custom rules for elaboration start (heating) and elaboration end (cooling).

Since all the automatic elaboration-related rules are an incredible mess, we have to put all the AST terms into this category one by one, and eliminate automatic elaboration heating/cooling rules altogether.
SYNTAX  \[ KItem ::= \text{customElabChildren}(KLabel) \] [function]

RULE

\[
\text{customElabChildren}(KL,KLabel) = \\
(KL = KLabel 'KListWrap) \lor (KL = KLabel 'Block) \lor (KL = KLabel 'For) \lor (KL = KLabel 'Catch) \lor (KL = KLabel 'LocalVarDecStm) \lor (KL = KLabel 'LocalVarDec) \lor (KL = KLabel 'SuperConstrInv) \lor (KL = KLabel 'ClassDecStm) \lor (KL = KLabel 'NewInstance) \lor (KL = KLabel 'QNewInstance)
\]

C.18  Module LITERALS

C.18.1  Auxiliary constructs

SYNTAX  \[ KItem ::= \text{hexToInt}(String) \] [function]

RULE

\[
\text{hexToInt}(Str:String) = \text{hexToInt}(0, Str)
\]

SYNTAX  \[ KItem ::= \text{hexToInt}(Int, String) \] [function]

RULE

\[
\text{hexToInt}(I:Int, Str:String) = \\
\text{hexToInt}(I, \text{retainHead}(Str, 1), \text{trimHead}(Str, 1))
\]

REQUIRES  \[ Str \neq \text{String} \]

RULE

\[
\text{hexToInt}(I:Int, "") = \text{I}
\]

SYNTAX  \[ KItem ::= \text{hexToInt}(Int, String, String) \] [function]

RULE

\[
\text{hexToInt}(I:Int, Digit:String, Str:String) = \\
\text{hexToInt}((I * \text{Int } 16) + \text{Int} \text{hexDigitToInt}(Digit), Str)
\]

SYNTAX  \[ KItem ::= \text{hexDigitToInt}(String) \] [function]

309
RULE

\[
\text{hexDigitToInt (Digit: String)}
\]
\[
\text{ordChar (Digit) } \leq \text{ordChar ("0")}
\]
\[
\text{REQUIRES ( ordChar (Digit) } \geq \text{ordChar ("0") } \land \text{Bool ( ordChar (Digit) } \leq \text{ordChar ("9") )}
\]

RULE

\[
\text{hexDigitToInt (Digit: String)}
\]
\[
\text{ordChar (Digit) } \leq \text{ordChar ("A") } + \text{Int 10}
\]
\[
\text{REQUIRES ( ordChar (Digit) } \geq \text{ordChar ("A") } \land \text{Bool ( ordChar (Digit) } \leq \text{ordChar ("F") )}
\]

RULE

\[
\text{hexDigitToInt (Digit: String)}
\]
\[
\text{ordChar (Digit) } \leq \text{ordChar ("a") } + \text{Int 10}
\]
\[
\text{REQUIRES ( ordChar (Digit) } \geq \text{ordChar ("a") } \land \text{Bool ( ordChar (Digit) } \leq \text{ordChar ("f") )}
\]

SYNTAX  \( KItem ::= \text{octaToInt ( String )} \) [function]

RULE

\[
\text{octaToInt (Str: String)}
\]
\[
\text{octaToInt (0, Str)}
\]

SYNTAX  \( KItem ::= \text{octaToInt ( Int, String )} \) [function]

RULE

\[
\text{octaToInt (I: Int, Str: String)}
\]
\[
\text{octaToInt (I, retainHead (Str, 1), trimHead (Str, 1))}
\]
\[
\text{REQUIRES Str } \neq \text{String ""}
\]

RULE

\[
\text{octaToInt (I: Int, "")}
\]
\[
\text{I}
\]

SYNTAX  \( KItem ::= \text{octaToInt ( Int, String, String )} \) [function]

RULE

\[
\text{octaToInt (I: Int, Digit: String, Str: String)}
\]
\[
\text{octaToInt ((I * Int 8) + Int octaDigitToInt (Digit), Str)}
\]
SYNTAX  
\[ KItem ::= \text{octaDigitToInt}(\text{String}) \] [function]

RULE
\[
\begin{align*}
\text{octaDigitToInt}(\text{Digit}:\text{String}) & \\
& \quad \text{hexDigitToInt}(\text{Digit})
\end{align*}
\]

SYNTAX  
\[ KItem ::= \text{octaAsciiToInt}(\text{Int}) \] [function]

RULE
\[
\begin{align*}
\text{octaAsciiToInt}(\text{I}:\text{Int}) & \\
& \quad \text{octaDigitToInt}(\text{chrChar(I)})
\end{align*}
\]

SYNTAX  
\[ KItem ::= \text{hexAsciiToInt}(\text{Int}) \] [function]

RULE
\[
\begin{align*}
\text{hexAsciiToInt}(\text{I}:\text{Int}) & \\
& \quad \text{hexDigitToInt}(\text{chrChar(I)})
\end{align*}
\]

C.18.2 Integer literals

RULE Lit-Deci
\[
\begin{align*}
\text{Lit('Deci}(	ext{Str:} \text{String'}) & \\
\text{'Lit('Deci}(	ext{Str:} \text{String'}) & \\
& \#\text{if } ( \text{lastChar} \text{(Str)} \text{=} \text{String }"i") \text{ V } \text{Bool} ( \text{lastChar} \text{(Str)} \text{=} \text{String }"L") \#\text{then } \text{String2Int} \text{(trimTail} \text{(Str,} 1\text{)}) :: \text{long } \#\text{else } \text{String2Int} \text{(Str)} :: \text{int } \#\text{fi}
\end{align*}
\]

RULE Lit-Hexa
\[
\begin{align*}
\text{Lit('Hexa}(	ext{Str:} \text{String'}) & \\
\text{'Lit('Hexa}(	ext{Str:} \text{String'}) & \\
& \#\text{if } ( \text{lastChar} \text{(Str)} \text{=} \text{String }"i") \text{ V } \text{Bool} ( \text{lastChar} \text{(Str)} \text{=} \text{String }"L") \#\text{then } \text{normalize} \text{(hexToInt} \text{(trimHead} \text{(trimTail} \text{(Str,} 1\text{),} 2\text{)}) :: \text{long }) \#\text{else } \text{normalize} \text{(hexToInt} \text{(trimHead} \text{(Str,} 2\text{)}) :: \text{int }) \#\text{fi}
\end{align*}
\]

RULE Lit-Octa
\[
\begin{align*}
\text{Lit('Octa}(	ext{Str:} \text{String'}) & \\
\text{'Lit('Octa}(	ext{Str:} \text{String'}) & \\
& \#\text{if } ( \text{lastChar} \text{(Str)} \text{=} \text{String }"i") \text{ V } \text{Bool} ( \text{lastChar} \text{(Str)} \text{=} \text{String }"L") \#\text{then } \text{normalize} \text{(octaToInt} \text{(trimHead} \text{(trimTail} \text{(Str,} 1\text{),} 1\text{)}) :: \text{long }) \#\text{else } \text{normalize} \text{(octaToInt} \text{(trimHead} \text{(Str,} 1\text{)}) :: \text{int }) \#\text{fi}
\end{align*}
\]
C.18.3 Float literals

RULE Lit-FFloat

\[ 'Lit('Float(Str:String))' \]

# if (lastChar(Str) == String "f") \lor Bool (lastChar(Str) == String "F") # then String2Float(trimTail(Str, 1)) :: float # else # if (lastChar(Str) == String "d") \lor Bool (lastChar(Str) == String "D") # then String2Float(trimTail(Str, 1)) :: double # else String2Float(Str) :: double # fi # fi

C.18.4 Boolean literals

RULE

\[ 'Lit('Bool('True('KList)))' \]

true

RULE

\[ 'Lit('Bool('False('KList)))' \]

false

C.18.5 Char literals

Chars are represented as int values, as described in Java specification.

RULE

\[ 'Lit('Char('Single(I:Int)))' \]

I :: char

RULE

\[ 'Lit('Char('NamedEscape(98)))' \]

8 :: char

RULE

\[ 'Lit('Char('NamedEscape(102)))' \]

12 :: char

RULE

\[ 'Lit('Char('NamedEscape(39)))' \]

39 :: char
RULE

'Lit('Char('NamedEscape(116))')
ordChar ('\t') :: char

RULE

'Lit('Char('NamedEscape(110))')
ordChar ('\n') :: char

RULE

'Lit('Char('NamedEscape(114))')
ordChar ('\r') :: char

RULE

'Lit('Char('NamedEscape(34))')
ordChar ('\"') :: char

RULE

'Lit('Char('OctaEscape1(I:Int))')
octaAsciiToInt (I) :: char

RULE

'Lit('Char('OctaEscape2(I1:I:Int, I2:I:Int))')
octaAsciiToInt (I1) * _Int 8 + _Int octaAsciiToInt (I2) :: char

RULE

octaAsciiToInt (I1) * _Int 64 + _Int octaAsciiToInt (I2) * _Int 8 + _Int octaAsciiToInt (I3) :: char

RULE

hexAsciiToInt (I1) * _Int 4096 + _Int hexAsciiToInt (I2) * _Int 256 + _Int hexAsciiToInt (I3) * _Int 16 + _Int hexAsciiToInt (I4) :: char
C.18.6 String literals

RULE

\[ \text{\prime Lit('String([ K1:K, K2:K, Ks:KList ]))} \]
\[ \text{plusAux ('Lit('String([ K1 ])), 'Lit('String([ K2, Ks ])))} \]

RULE

\[ \text{\prime Lit('String([ 'Chars(Str:String ) ]))} \]
\[ \text{Str} \]

RULE

\[ \text{\prime Lit('String([ K.K ])}} \]
\[ \text{charToString ('Lit('Char(K)))} \]
\[ \text{REQUIRES getKLabel ( K ) \#KLabel 'Chars} \]

RULE

\[ \text{\prime Lit('String([ 'KList ]))} \]
\[ \text{""} \]

SYNTAX  \[ KItem ::= \text{charToString ( K ) [strict]} \]

RULE

\[ \text{charToString (I:Int :: char )} \]
\[ \text{chrChar (I)} \]

C.18.7 Null literal

RULE

\[ \text{\prime Lit('Null(KList))} \]
\[ \text{null} \]
C.19 Module PROCESS-LOCAL-CLASSES

C.19.1 Main rules

\begin{verbatim}
rule elab-ClassDecStm
  elab ('ClassDecStm(ClassDec::K))
  processLocalClassDec (ClassDec, generateLocalClassType (ClassDec), false)
\end{verbatim}

generates a ClassType for the given 'ClassDec term representing a local class

\textbf{Syntax} \quad KItem ::= generateLocalClassType (K)

\begin{verbatim}
rule generateLocalClassType
  generateLocalClassType ('ClassDec('ClassDecHead([OldModifiers::KList
    , ClassName::Id, ...
    , -:K, -:K, -:K, -:K]))
  getClassType (toPackage (CrntClass), String2Id (Id2String (ClassName)
  + String "$" + String Int2String (LocalId)))
\end{verbatim}

\(\langle\langle \text{CrntClass} : \text{ClassType}\rangle\rangle\)

\(\text{crntClass}\)

\begin{verbatim}
nextLocalId\)
\end{verbatim}

Register a local class in the given generated package and fully process it.

\textbf{Syntax} \quad KItem ::= processLocalClassDec (K, ClassType, Bool) [strict(2)]

\begin{verbatim}
rule processLocalClassDec-register-name
  processLocalClassDec ('ClassDec('ClassDecHead(
    [ OldModifiers::KList ]
    , generateLocalClassModifiers (CT, [ OldModifiers ])
    , InitName::Id, ...
    , ClassName::Id, ...
    , getSimpleName (GeneratedClass)
    , -:K, -:K, -:K),
     GeneratedClass::ClassType, false)
  mapWrap (LocalTypes:Map LocalTypes [ GeneratedClass / InitName ])
\end{verbatim}

\(\text{LocalTypes}\)

The second argument: existing modifiers

\textbf{Syntax} \quad KListWrap ::= generateLocalClassModifiers (ContextType, KListWrap) [function]

315
RULE
generateLocalClassModifiers (CT:ContextType, [OldModifiers:KList])

#if CT =_K static #then [‘Static(KList), OldModifiers] #else [OldModifiers] #fi

RULE PROCESSLOCALCLASSDEC-PROCESS

processLocalClassDec (ClassDec:K, Class:ClassType, true)

registerClass (ClassDec, toPackage (CrntClass), Class, CrntClass)

↷ localClassTransformIntoInner (Class, localClassGetAccessibleOuterEnv (Class)) ↷ loadElabBuffer

⟨ CrntClass:ClassType ⟩crntClass

Passes the freshly discovered/constructed class declaration through all the preprocessing steps except elaboration. Only usable for individual classes, when the rest of the classpath is already preprocessed.

SYNTAX  KItem ::= registerClass (K, PackageId, ClassType, ClassType)

RULE registerCLASS

registerClass (ClassDec:K, Pack:PackageId, Class:ClassType, EnclosingClass:ClassType)

processTypeNames ([ClassDec], Pack) ↷ processTypeDecsInPCUPhase ([ClassDec], Pack, EnclosingClass) ↷ processClassDecs (setWrap (Class))

↷ processTypeWithDepends (Class)

Computes the accessible outer local environment for the given local class. {Accessible outer env} = {the whole outer env} - {all names X that could be resolved by elabLookup(Class, X)} elabLookup don’t search the var in outer classes. Just in this class and base classes.

SYNTAX  KItem ::= localClassGetAccessibleOuterEnv (ClassType)

| localClassGetAccessibleOuterEnv (ClassType, Map, Map, Id, K) [strict(5)]

RULE LOCALCLASSGETACCESSIBLEOUTERENV-START

localClassGetAccessibleOuterEnv (Class:ClassType)

localClassGetAccessibleOuterEnv (Class, EnclosingLocalEnv, ‘Map, ‘K, ‘K)

⟨ mapWrap (EnclosingLocalEnv:Map) ⟩elabEnv

RULE LOCALCLASSGETACCESSIBLEOUTERENV-START-TEST

localClassGetAccessibleOuterEnv (Class:ClassType, X:Id -> T:Type)

‘Map — Map — Map

‘K — X

elabLookup (X, Class)
rule localClassGetAccessibleOuterEnv-test-found

\[
\text{localClassGetAccessibleOuterEnv (Class:ClassType, } X:Id \mapsto \text{Map, } X:Id, \\
\text{KR:KResult}) \]

\[\xrightarrow{\ K \ }\]

REQUIRES \( \text{KR} \neq K \cdot K \)

rule localClassGetAccessibleOuterEnv-end

\[
\text{localClassGetAccessibleOuterEnv (Class:ClassType, } \text{mapWrap(AccessibleLocalEnv:Map), } \text{NewOuterEnv:Map, } \xrightarrow{\ K \ })
\]

\[
\text{mapWrap (NewOuterEnv)}
\]

Convert the given local class into an equivalent inner class, with the following transformations:

- Add a field to the class for each outer local variable
- For each constructor add an argument for each outer local variable

SYNTAX \( KItem ::= \text{localClassTransformIntoInner (ClassType, K)} \) [strict(2)]

rule localClassTransformIntoInner

\[
\text{localClassTransformIntoInner (Class:ClassType, mapWrap (AccessibleLocalEnv:Map))} \\
\text{#if } \sim\text{Bool isEmpty (AccessibleLocalEnv)} =_K \text{true } \#\text{then localClassTransformIntoInnerImpl (Class:ClassType, mapWrap (AccessibleLocalEnv:Map, generateLocalEnvVarName (generateLocalEnvClassType (Class)) \mapsto generateLocalEnvClassType (Class))) } \#\text{else } K \#\text{fi}
\]

Same as localClassTransformIntoInner but AccessibleLocalEnv has one more entry — the field Local-Class@LocalEnv_obj that will be created later, a self reference required when a local class instantiates either itself or an enclosing local class. See tests 957, 958.

SYNTAX \( KItem ::= \text{localClassTransformIntoInnerImpl (ClassType, K)} \)

rule localClassTransformIntoInnerImpl

\[
\text{localClassTransformIntoInnerImpl (Class:ClassType, mapWrap (AccessibleLocalEnv:Map))} \\
\sim\text{localEnvObjectBuild (generateLocalEnvClasstype (Class), AccessibleLocalEnv)} \\
\sim\text{localClassAugmentFields (Class, generateLocalEnvClassType (Class))} \\
\sim\text{localClassAugmentConsParams (Class, getConsName (Class), generateLocalEnvClassType (Class))}
\]

SYNTAX \( KItem ::= \text{generateLocalEnvClasstype (ClassType)} \) [function]
RULE

generateLocalEnvClassType ( class X:Id)

class String2Id ( Id2String (X) + String "$LocalEnv")

Add the given fields definitions to the given class. The class is in MembersProcessedCPhase

SYNTAX  KItem ::= localEnvObjectBuild ( K, Map ) [strict(1)]

RULE localEnvObjectBuild

localEnvObjectBuild ( LocalEnvClass:ClassType, AccessibleLocalEnv:Map )

\begin{align*}
\text{localEnvClassAddFieldsAndRegister} & ( \langle \text{ClassDec}(\langle \text{ClassDecHead}(\langle \langle CT:ContextType \rangle \rangle) ) \rangle \rangle, \langle \text{ClassBody}(\langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \associ
Add the given fields definitions to the given class. The class is in MembersProcessedCPhase
RULE localClassAugmentConsParams-end

\[
\text{localClassAugmentConsParams (LocalClass: } \text{ClassType}, \text{ConsName: Id, LocalEnvClass: } \text{ClassType}) \rightarrow \kappa
\]

\[
\text{LocalClass} \text{classType } \text{Methods: } \text{Map methods}
\]

REQUIRES \( \neg \text{Bool haveUnaugmentedConstructors (ConsName, LocalEnvClass, Methods) } =_\kappa \text{ true} \)

SYNTAX \( Kitem ::= \text{haveUnaugmentedConstructors (Id, ClassType, Map) [function]} \)

RULE haveUnaugmentedConstructors (ConsName: Id, LocalEnvClass: ClassType, ( ( (sig (ConsName: Id, types (TList: KList)) ) \mapsto \ldots ) \text{Map})

REQUIRES \( \neg \text{Bool getLastKListElement ([TList] } ) =_\kappa \text{ LocalEnvClass} \)

Elaborates the given node and saves it to <elabBuffer>

SYNTAX \( Kitem ::= \text{appendToElabBuffer (K)} \)

CONTEXT

\[
\text{appendToElabBuffer (} \square \text{)}
\]

\[
[\text{result(ElabKResult)}]
\]

RULE appendToElabBuffer

\[
\text{appendToElabBuffer (elabRes(K:K)) } \rightarrow \kappa \text{ [ } \text{elabBuffer} \text{[KList:K]} \text{]}
\]

SYNTAX \( Kitem ::= \text{loadElabBuffer} \)

RULE loadElabBuffer

\[
\text{loadElabBuffer } \rightarrow \kappa \text{ [ElabBuffer:K]} \text{elabBuffer}
\]

C.19.2 Auxiliary functions for other modules

SYNTAX \( Kitem ::= \text{localClassGetExtraArgs (ClassType)} \)

When <enclosingLocalEnv> is not empty (only possible for certain local classes), we add one more argument to the constructor call — the local environment object.
C.20 Module PROCESS-ANONYMOUS-CLASSES

RULE elab-NEWINSTANCE-anonymous-implements-interface

\[
\text{elab} (\text{'NewInstance(Arg1:K, Class:ClassType, elabRes ( [ ActualArgs:KList ]), }'\text{Some('ClassBody([ AnonClassDecs:KList ]))})
\]
\[
\text{processAnonymousClass ( String2Id ("Anonymous_$" + String2Id (LocalId)), class Object, [ Class ], Arg1, [ ActualArgs ], [ AnonClassDecs ], kr[ 'KList ], kr[ 'KList ])}
\]
\[
\text{LocalId:} \text{Int} \quad \text{nextLocalId} \quad \text{Class:ClassType} \quad \text{interface:} \text{classMetaType}
\]

RULE elab-NEWINSTANCE-anonymous-extends-class

\[
\text{elab} (\text{'NewInstance(Arg1:K, Class:ClassType, elabRes ( [ ActualArgs:KList ]), }'\text{Some('ClassBody([ AnonClassDecs:KList ]))})
\]
\[
\text{processAnonymousClass ( String2Id ("Anonymous_$" + String2Id (LocalId)), Class, [ 'KList ], Arg1, [ ActualArgs ], [ AnonClassDecs ], buildConstructorFormalParams ( [ ActualArgs ], [ 'KList ], 0), buildConstructorFirstLineArgs ( [ ActualArgs ], [ 'KList ], 0))}
\]
\[
\text{LocalId:} \text{Int} \quad \text{nextLocalId} \quad \text{Class:ClassType} \quad \text{class:} \text{classMetaType}
\]

SYNTAX  

\[ KItem ::= \text{processAnonymousClass ( Id, ClassType, KListWrap, K, KListWrap, KListWrap, K, K ) [strict(7,8)]} \]

In the current approach stmtAndExp cannot be eliminated from here. The local class declaration will be transformed into the instantiation and initialization of the parameter object. This will imply a 'NewInstance and zero or more 'Assign statements. Theoretically for anonymous classes the parameter object could be instantiated right in the 'NewInstance expression of its local class, with parameters passed as constructor arguments, thus eliminating the need of a separate variable declaration. But this would complicate the semantics too much. So in theory, we could eliminate stmtAndExp even from this place.
RULE processAnonymousClass
processAnonymousClass (Name:Id, BaseClass:ClassType, [ BaseInterfaces:KList ], Arg1:K, [ ActualArgs:KList ], [ AnonClassDecs:KList ], kr[ ConstructorFormalParams:KList ],
kr[ ConstructorFirstLineArgs:KList ])
elab (stmtAndExp ('ClassDecStm('ClassDec('ClassDecHead('KList '), Name, 'None('KList ), 'Some('SuperDec(BaseClass)'), 'Some('ImplementsDec('BaseInterfaces))))), 'ClassBody('
[ AnonClassDecs, 'ConstrDec('ConstrDecHead('Public('KList ), 'None('KList ), Name, [ ConstructorFormalParams ], 'None('KList ), 'ConstrBody('Some('SuperConstrInv('None('KList ), [ ConstructorFirstLineArgs ]))), [ 'KList ]))), 'NewInstance(Arg1, 'TypeName(Name), elabRes ([ ActualArgs ], 'None('KList )))}

Build the part of the constructor of an anonymous class definition corresponding to formal params, and arguments of the superclass constructor invocation, respectively

- First KList — list of actual arguments
- Second KList — the result
- Third arg — a counter used to generate var names

SYNTAX KItem ::= buildConstructorFormalParams (KListWrap, KListWrap, Int) | buildConstructorFirstLineArgs (KListWrap, KListWrap, Int)

RULE
buildConstructorFormalParams ( [ActualParam:K -> 'KList, --, [ --, 'KList
Param([ 'KList ], typeof (ActualParam), String2Id ('$' + String2Int2String (nu m)))

nu m: Int

nu m + Int 1

RULE
buildConstructorFormalParams ( [ 'KList ], [ Ks:KList ], --)

kr[ Ks ]

RULE
buildConstructorFirstLineArgs ( [ --:K -> 'KList, --, [ --,
'KList
ExprName(String2Id ('$' + String2Int2String (nu m)))

nu m: Int

nu m + Int 1

322
RULE

buildConstructorFirstLineArgs ( [ 'KList ], [ Ks:KList ], — )

kr[ Ks ]

RULE elab-QNewInstance-anonymous-extends-class

elab ( 'QNewInstance(elabRes (TypedQual:K), elabRes (Arg2:K),
Class:ClassType, elabRes (Arg4:K), elabRes ( [ ActualArgs:KList ] ), 'Some(
'ClassBody([ AnonClassDecs:KList ] )))

processQAnonymousClass ( String2Id ("Anonymous_" + String Int2String (LocalId)
), Class, TypedQual, Arg2, [ ActualArgs ], [ AnonClassDecs ],
buildConstructorFormalParams ( [ TypedQual, ActualArgs ], [ 'KList ], 0),
builtConstructorFirstLineArgs ( [ TypedQual, ActualArgs ], [ 'KList ], 0))

LocalId:int

LocalId + int 1

nextLocalId ( Class ) classType ( class ) classMetaType

SYNTAX

KItem ::= processQAnonymousClass ( Id, ClassType, K, K, KListWrap, KListWrap, K, K ) [strict(7,8)]

RULE processQAnonymousClass

processQAnonymousClass ( Name:Id, BaseClass:ClassType, TypedQual:K, Arg2:K,
[ ActualArgs:KList ], [ AnonClassDecs:KList ], kr[ ConstructorFormalParams:KList ],


elab ( stmtAndExp ( 'ClassDecStm('ClassDec('ClassDecHead([ 'KList ], Name, 'None('KList),

'Some('SuperDec(BaseClass)), 'Some('ImplementsDec([ 'KList ] ))), 'ClassBody(

[ AnonClassDecs, 'ConstrDec('ConstrDecHead([ 'Public('KList ] ), 'None('KList ), Name,

[ ConstructorFormalParams ], 'None('KList ), 'ConstrBody('Some('QSuperConstrInv(

ConstrQual, 'None('KList ), [ ConstructorFirstLineArgs ] ), [ 'KList ] ))) )]),

'NewInstance(Arg2, 'TypeName(Name), elabRes ( [ TypedQual, ActualArgs ] ), 'None(

'KList ))))

C.21 Module FOLDING

RULE FoldingPhase-start

foldingPhase k ElaborationPhase globalPhase Bag

The sole content of <k> cell during FoldingPhase

SYNTAX

KItem ::= foldingPhase

323
RULE FOLDING-START

\[
\langle \text{foldingPhase} \rangle_k \langle \text{Class:ClassType} \rangle_{\text{classType}} \langle \text{MetaType:ClassMetaType} \rangle_{\text{classMetaType}}
\]

\[
\langle \text{AccessMode:AccessMode} \rangle_{\text{classAccessMode}} \langle \text{BaseClass:ClassType} \rangle_{\text{extends}}
\]

\[
\langle [ \text{InstanceFields:KList} ] \rangle_{\text{instanceFields}} \langle [ \text{StaticFields:KList} ] \rangle_{\text{staticFields}}
\]

\[
\langle [ \text{StaticInit:KList} ] \rangle_{\text{staticInit}}
\]

\[
\text{getClassDecLabel}(\text{MetaType})\langle \text{ClassDecHead}(\text{AccessMode}, \text{Class}, \text{'None'}(\text{KList}), \text{'Some'}(\text{SuperDec}(\text{BaseClass})), \text{'Some'}(\text{ImplementsDec}([\text{'KList'}]))), \text{'ClassBody'}([\text{StaticFields}, \text{'StaticInit'}(\text{Block}(\text{StaticInit})), \text{InstanceFields}]))
\]

SYNTAX  \( KLabel ::= \text{getClassDecLabel}(\text{ClassMetaType}) \) [function]

RULE

\[
\text{getClassDecLabel}(\text{class})
\]

\( '\text{ClassDec} \)

RULE

\[
\text{getClassDecLabel}(\text{interface})
\]

\( '\text{InterfaceDec} \)

RULE FOLDING-IMPLEMENTSDEC

\[
\langle \text{foldingPhase} \rangle_k \langle \text{Class:ClassType} \rangle_{\text{classType}} \langle \text{BaseInterface:ClassType} \rangle_{\text{implTrans}}
\]

\[
\langle \text{KLabel}(\text{ClassDecHead}(\text{K}, \text{K}, \text{K}, \text{K}, \text{'Some'}(\text{ImplementsDec}([\text{'KList'}]))), \text{ClassBody}(\text{BaseInterface})) \rangle
\]

folded
RULE FOLDING-METHODDEC

\[
\begin{align*}
\text{foldingPhase}_k \text{ Class:ClassType classType} \\
\text{sig (Name:Id, —)} \text{ methodSignature} \quad \text{ReturnType:Type methodReturnType} \\
\text{[ Params:KList ] methodParams \quad MethodBody:K methodBody} \\
\text{Acc:AccessMode methodAccessMode \quad CT:ContextType methodContextType} \\
\end{align*}
\]

\[
\begin{align*}
\text{Bag} \\
\text{ClassDecHead(—), ClassBody([—,} \\
\text{MethodDec( MethodDecHead( Acc, CT ], None(KList), ReturnType,} \\
\text{Name, [ Params ], None(KList), MethodBody) }
\end{align*}
\]

RULE FOLDING-CLASS-END

\[
\begin{align*}
\text{foldingPhase}_k \text{ Class:ClassType classType} \\
\text{Bag implTrans Bag} \\
\text{classPhase}
\end{align*}
\]

RULE FOLDING-CLASS-FOLD-TOP-LEVEL

\[
\begin{align*}
\text{foldingPhase}_k \text{ ClassPack:PackageId \mapWrap Map \namesToClasses} \\
\text{Class:ClassType classType} \quad \text{enclosingClass ClassDec:K class} \\
\text{FoldedCPhase classPhase} \\
\text{Bag}
\end{align*}
\]

\[
\begin{align*}
\text{[—,} \\
\text{ClassDec program}
\end{align*}
\]

REQUIRES ClassPack =_K toPackage (Class)
RULE FOLDING-CLASS-FOLD-INNER

\[ \langle foldingPhase \rangle_k (\text{ClassPack: PackageId} \rightarrow \text{mapWrap} (\text{Map})) \]
\[ \langle EnclosingClassPack: PackageId \rightarrow \text{mapWrap} (\text{Map} \rightarrow \text{Class}) \rangle \rightarrow \text{namesToClasses} \]
\[ \langle \text{Class: ClassType} \rangle_{\text{classType}} (\text{EnclosingClass: ClassType} \rightarrow \text{enclosingClass} \rightarrow \text{class}) \]
\[ \langle \text{ClassDec: K} \rangle_{\text{folded}} (\text{FoldedCPhase} \rightarrow \text{classPhase}) \]

\[ \langle \text{EnclosingClass} \rangle_{\text{classType}} \]
\[ \langle \text{FoldedCPhase} \rangle_{\text{classPhase}} \]

\[ \langle \text{EnclosingClass} \rangle_{\text{classType}} (\text{ClassBody}(\text{KList})) \rightarrow \text{folded} \rightarrow \text{class} \]

\[ \langle \text{EnclosingClass} \rangle_{\text{classType}} (\text{ClassBody}(\text{KList})) \rightarrow \text{folded} \rightarrow \text{class} \]

\[ \text{REQUIRES ClassPack} = K \text{toPackage (Class)} \land \text{Bool EnclosingClassPack} = K \text{toPackage (EnclosingClass)} \]

RULE FOLDINGPHASE-END

\[ \langle foldingPhase \rangle_k (\text{FoldingPhase} \rightarrow \text{globalPhase}) \rightarrow \text{classes} \]
Appendix D
K-Java Syntax

This chapter contains the entire syntax of Java 5. This syntax was developed by Shijiao Yuwen from Formal Systems Laboratory of UIUC. It is an adaptation of Java-Front [12], the same syntax definition used to produce the external parser for K-Java. Although the syntax is fairly complete, it might still contain errors. This is because at the moment of writing it was not yet used to produce a parser for Java, consequently it was not yet used to parse programs. Also, only a small part of K-Java is now defined over Java syntax. The rest of the semantics is still in AST form. Thus, most of this syntax was not put to use yet. Yet, the part of this syntax that is used is expressions, statements, and for the rest of the syntax, the attributes (strictness and klabel). Syntax definition was added to K-Java at a later stage of development, when the semantics for Java 1.4 was already complete.

D.1 Module EXP-SYNTAX

```
SYNTAX  Exp ::= StmtExp

SYNTAX  StmtExp ::= PrefixPostfixExp
               | AssignExp
               | MethodInvokeExp
               | ClassInstanceCreationExp

MethodInvocation

SYNTAX  MethodInvokeExp ::= MethodSpec ( Exps ) [klabel('Invoke)]

SYNTAX  MethodSpec ::= MethodName [klabel('Method)]
               | Exp . OptionalTypeArgs Id [klabel('Method)]
               | super . OptionalTypeArgs Id [klabel('SuperMethod)]
               | TypeName . super . OptionalTypeArgs Id [klabel('QSuperMethod)]
               | AmbName . TypeArgs Id [klabel('GenericMethod)]

ClassInstanceCreation

SYNTAX  ClassInstanceCreationExp ::=new OptionalTypeArgs ClassOrInterfaceType ( Exps )
                                           OptionalClassBody [klabel('NewInstance)]
```
AssignmentOperators

Syntax: LHS ::= ExprName
| FieldAccess
| ArrayAccess

Syntax: Exp ::= LHS

Syntax: AssignExp ::= LHS = Exp [klabel('Assign')]

Syntax: AssignExp ::= CompoundAssignExp

Syntax: CompoundAssignExp ::= LHS *= Exp [klabel('AssignMul')]
| LHS /= Exp [klabel('AssignDiv')]
| LHS %= Exp [klabel('AssignRemain')]
| LHS += Exp [klabel('AssignPlus')]
| LHS -= Exp [klabel('AssignMinus')]
| LHS <<= Exp [klabel('AssignLeftShift')]
| LHS >>= Exp [klabel('AssignRightShift')]
| LHS &= Exp [klabel('AssignAnd')]
| LHS ^= Exp [klabel('AssignExcOr')]
| LHS |= Exp [klabel('AssignOr')]

UnaryOperators

Syntax: Exp ::= (PrimType) Exp [klabel('CastPrim')]

Syntax: Exp ::= (RefType) Exp [klabel('CastRef')]

Syntax: Exp ::=˜ Exp [strict, klabel('Complement')]
| ! Exp [strict, klabel('Not')]
| + Exp [strict, klabel('Plus')]
| - Exp [strict, klabel('Minus')]

Syntax: PrefixPostfixExp ::=++ Exp [klabel('PreIncr')]
| -- Exp [klabel('PreDecr')]
Postfix

**syntax**  
\[
\text{PrefixPostfixExp ::= Exp ++ [klabel('PostIncr)]} \\
| Exp -- [klabel('PostDecr)]
\]

Binary Operators

**syntax**  
\[
\text{Exp ::= Exp * Exp [seqstrict, klabel('Mul)]} \\
| Exp / Exp [seqstrict, klabel('Div)] \\
| Exp \% Exp [seqstrict, klabel('Remain)] \\
| Exp + Exp [seqstrict, klabel('Plus)] \\
| Exp - Exp [seqstrict, klabel('Minus)]
\]

**syntax**  
\[
\text{Exp ::= Exp << Exp [seqstrict, klabel('LeftShift)]} \\
| Exp >> Exp [seqstrict, klabel('RightShift)] \\
| Exp >>> Exp [seqstrict, klabel('URightShift)]
\]

**syntax**  
\[
\text{Exp ::= Exp > Exp [seqstrict, klabel('Gt)]}
\]

**syntax**  
\[
\text{Exp ::= Exp < Exp [seqstrict, klabel('Lt)]}
\]

**syntax**  
\[
\text{Exp ::= Exp >= Exp [seqstrict, klabel('GtEq)]}
\]

**syntax**  
\[
\text{Exp ::= Exp || Exp [strict(1), klabel('LazyOr)]} \\
| Exp && Exp [strict(1), klabel('LazyAnd)] \\
| Exp | Exp [seqstrict, klabel('Or)] \\
| Exp ^ Exp [seqstrict, klabel('ExcOr)] \\
| Exp & Exp [seqstrict, klabel('And)]
\]

**syntax**  
\[
\text{Exp ::= Exp ? Exp : Exp [klabel('Cond)]}
\]

**syntax**  
\[
\text{Exp ::= Exp instanceof RefType [strict, klabel('InstanceOf)]}
\]

Field Access

**syntax**  
\[
\text{FieldAccess ::= Exp . Id [klabel('Field)]} \\
| super . Id [klabel('SuperField)] \\
| TypeName . super . Id [klabel('QSuperField)]
\]

Array Access

**syntax**  
\[
\text{ArrayAccess ::= Exp [ Exp ] [seqstrict, klabel('ArrayAccess)]}
\]

329
ArrayCreation

SYNTAX   Exp ::= ArrayCreationExp

SYNTAX   ArrayCreationExp ::= new Type DimExps Dims  [strict(2), klabel('NewArray)]

SYNTAX   ArrayCreationExp ::= new Type Dims ArrayInit  [klabel('NewArray)]

SYNTAX   ArrayBaseType ::= PrimType
   |  TypeName
   |  TypeName < ? > [klabel('UnboundWld)]

SYNTAX   Dim ::= [ ]  [klabel('Dim)]

SYNTAX   DimExp ::= [ Exp ]  [strict, hybrid, klabel('Dim)]

ArrayInitializers

SYNTAX   ArrayInit ::= { VarInits }  [klabel('ArrayInit)]

SYNTAX   ArrayInit ::= { VarInits , }  [klabel('ArrayInit)]

Primary

SYNTAX   Exp ::= Literal  [klabel('Lit)]

SYNTAX   Literal ::= IntLiteral
   |  FloatLiteral
   |  BoolLiteral
   |  CharLiteral
   |  StringLiteral
   |  NullLiteral
   |  ClassLiteral

SYNTAX   ClassLiteral ::= Type . class  [klabel('Class)]
   |  void . class  [klabel('VoidClass)]

SYNTAX   Exp ::= this  [klabel('This)]
   |  TypeName . this  [klabel('QThis)]

SYNTAX   Exp ::= ( Exp )  [bracket]
D.2 Module STMT-SYNTAX

Blocks

SYNTAX  BlockStmt ::= Stmt
            | LocalVarDecStmt
            | ClassDec  [klabel('ClassDecStm')]

SYNTAX  Block ::= { K } [klabel('Block')]

LocalVariableDeclarations

SYNTAX  LocalVarDecStm ::= LocalVarDec  ;  [prefer, klabel('LocalVarDecStm')]

SYNTAX  LocalVarDec ::= AnnoVarModList Type VarDecList  [prefer, klabel('LocalVarDec')]

Statements

SYNTAX  Stmt ::= StmtWithoutTrailing
            | LabeledStmt
            | IfThenElseStmt
            | IfThenStmt
            | WhileStmt
            | ForStmt

SYNTAX  StmtWithoutTrailing ::= Block
            | EmptyStmt
            | ExprStmt
            | AssertStmt
            | SwitchStmt
            | DoStmt
            | TryStmt
            | StackConsumerStmt
            | SynchronizedStmt
            | ThrowStmt

SYNTAX  StackConsumerStmt ::= ContinueStmt
            | BreakStmt
            | ReturnStmt
IfThenElseStmt ::= if ( Exp ) Stmt else Stmt [strict(1), klabel('If')]

IfThenStmt ::= if ( Exp ) Stmt [prefer, klabel('If')]

WhileStmt ::= while ( Exp ) Stmt [klabel('While')]

ForStmt ::= for ( LocalVarDec ; OptionalExp ; Exps ) Stmt [klabel('For')]

ForStmt ::= for ( Exps ; OptionalExp ; Exps ) Stmt [klabel('For')]

ForStmt ::= for ( Param : Exp ) Stmt [klabel('ForEach')]

LabeledStmt ::= Id : Stmt [klabel('Labeled')]

EmptyStmt ::= ; [klabel('Empty')]

ExprStmt ::= Exp ; [strict, klabel('ExprStm')]

AssertStmt ::= assert Exp ; [strict, klabel('AssertStm')]
| assert Exp : Exp ; [strict(1), klabel('AssertStm')]

SwitchStmt ::= switch ( Exp ) SwitchBlock [strict(1), klabel('Switch')]

SwitchBlock ::= { SwitchGroupList SwitchLabelList } [klabel('SwitchBlock')]

SwitchGroup ::= SwitchLabelList BlockStmList [klabel('SwitchGroup')]

SwitchLabel ::= case Exp : [klabel('Case')]
| default : [klabel('Default')]

DoStmt ::= do Stmt while ( Exp ) ; [strict(2), klabel('DoWhile')]

CatchClause ::= catch ( Param ) Block [klabel('Catch')]

TryStmt ::= try Block CatchClauses finally Block [klabel('Try')]
| try Block CatchClauses [klabel('Try')]

ThrowStmt ::= throw Exp ; [strict, klabel('Throw')]

ContinueStmt ::= continue OptionalId ; [klabel('Continue')]

BreakStmt ::= break OptionalId ; [klabel('Break')]
SYNTAX  ReturnStmt ::= return OptionalExp ; [klabel('Return)]

SYNTAX  SynchronizedStmt ::= synchronized ( Exp ) Block [strict(1), klabel('Synchronized)]

D.3 Module TYPE-SYNTAX

SYNTAX  Type ::= PrimType
          | RefType
          | void [klabel('Void)]

ParameterizedTypes

SYNTAX  TypeArgs ::=< ActualTypeArgList > [klabel('TypeArgs)]

SYNTAX  ActualTypeArg ::= Type
          | ? OptionalWildcardBound [klabel('Wildcard')]

SYNTAX  WildcardBound ::= extends RefType [klabel('WildcardUpperBound')]
          | super RefType [klabel('WildcardLowerBound')]

PrimitiveTypes

SYNTAX  IntOrLongType ::= int [klabel('Int')]
          | long [klabel('Long')]

SYNTAX  IntType ::= byte [klabel('Byte')]
          | short [klabel('Short')]
          | char [klabel('Char')]
          | IntOrLongType

SYNTAX  FloatType ::= float [klabel('Float')]
          | double [klabel('Double')]

SYNTAX  NumericType ::= IntType
          | FloatType

SYNTAX  PrimType ::= NumericType
          | bool [klabel('Boolean')]

ReferenceTypes

SYNTAX  ClassOrInterfaceType ::= TypeDecSpec OptionalTypeArgs [klabel('ClassOrInterfaceType)]
### D.4 Module CLASS-SYNTAX

#### ClassDeclarations

**Syntax**

\[
\text{ClassDecHead ::= AnnoClassModList \ class} \\
\quad \text{Id} \ 	ext{OptionalTypeParams} \ 	ext{OptionalSuper} \ 	ext{OptionalInterfaces} \\
\quad \text{[klabel('ClassDecHead)]}
\]

\[
\text{ClassBody ::= \{ ClassBodyDecList \} \ [klabel('ClassBody)]}
\]

\[
\text{ClassDec ::= ClassDecHead ClassBody \ [klabel('ClassDec)]}
\]

\[
\text{ClassDec ::= EnumDec}
\]
SYNTAX  
ClassMod ::= Public
         | Private
         | Protected
         | Abstract
         | Final
         | Static
         | StrictFP

SYNTAX  
AnnoClassMod ::= Anno
         | ClassMod

SYNTAX  
Super ::= extends JavaClassType [klabel('SuperDec)]

SYNTAX  
Interfaces ::= implements InterfaceTypeList [klabel('ImplementsDec)]

SYNTAX  
ClassBodyDec ::= InstanceInit
         | StaticInit
         | ClassMemberDec
         | ConstrDec

SYNTAX  
ClassMemberDec ::= FieldDec
         | ClassDec
         | MethodDec
         | InterfaceDec
         | SemiColon

SYNTAX  
SemiColon ::= ; [klabel('SemiColon)]

ConstructorDeclarations

SYNTAX  
ConstrHead ::= AnnoConstrModList OptionalTypeParams Type Id( Params ) OptionalThrows
             [klabel('ConstrDecHead)]

SYNTAX  
ConstrBody ::= { OptionalConstrInv BlockStmtList } [klabel('ConstrBody)]

SYNTAX  
ConstrDec ::= ConstrHead ConstrBody [klabel('ConstrDec)]

SYNTAX  
ConstrInv ::= OptionalTypeArgs this ( Exps ) ; [klabel('AltConstrInv)]

SYNTAX  
ConstrInv ::= OptionalTypeArgs super ( Exps ) ; [klabel('SuperConstrInv)]

SYNTAX  
ConstrInv ::= Exp . OptionalTypeArgs super ( Exps ) ; [klabel('QSuperConstrInv)]
**SYNTAX**

\[\text{ConstrMod ::= Public} \]
\[\text{ Private} \]
\[\text{ Protected} \]

**SYNTAX**

\[\text{AnnoConstrMod ::= Anno} \]
\[\text{ ConstrMod} \]

**EnumDeclarations**

**SYNTAX**

\[\text{EnumDecHead ::= AnnoClassModList enum Id OptionalInterfaces} \]
\[\text{ EnumBody ::= \{} \text{EnumConstList OptionalEnumBodyDecs} \} \]
\[\text{ EnumDec ::= EnumDecHead EnumBody} \]

**FieldDeclarations**

**SYNTAX**

\[\text{FieldDec ::= AnnoFieldModList Type VarDecList} \]
\[\text{ VarDec ::= VarDecId} \]
\[\text{ VarDecId ::= Id} \]
\[\text{ VarInit} \]
\[\text{ FieldMod ::= Public} \]
\[\text{ Private} \]
\[\text{ Protected} \]
\[\text{ Final} \]
\[\text{ Static} \]
\[\text{ Transient} \]
\[\text{ Volatile} \]
SYNTAX  AnnoFieldMod ::= Anno
        | FieldMod

MethodDeclarations

SYNTAX  MethodDecHead ::= AnnoMethodModList OptionalTypeParams Type Id ( Params )
                   OptionalThrows [klabel('MethodDecHead')]

SYNTAX  MethodBody ::= Block
        | ; [klabel('NoMethodBody')]

SYNTAX  MethodDec ::= MethodDecHead MethodBody  [klabel('MethodDec')]

SYNTAX  Param ::= AnnoVarModList Type VarDecId  [klabel('Param')]

SYNTAX  Param ::= AnnoVarModList Type  ...  VarDecId  [klabel('VarArityParam')]

SYNTAX  Throws ::= throws ExceptionTypeList  [klabel('ThrowsDec')]

SYNTAX  ExceptionType ::= JavaClassType

SYNTAX  MethodMod ::= Public
        | Private
        | Protected
        | Abstract
        | Final
        | Static
        | Native
        | Synchronized
        | StrictFP

SYNTAX  AnnoMethodMod ::= Anno
        | MethodMod

SYNTAX  VarMod ::= Final

SYNTAX  AnnoVarMod ::= Anno
        | VarMod

InstanceInitializers

SYNTAX  InstanceInit ::= Block  [klabel('InstanceInit')]
StaticInitializers

**SYNTAX**  \( \text{StaticInit ::= static Block} \)  \([\text{klabel('StaticInit')}]\)

## D.5 Module INTERFACE-SYNTAX

### AbstractMethodDeclarations

**SYNTAX**  \( \text{AbstractMethodDec ::= AnnoAbstractMethodModList OptionalTypeParams Type Id ( Params ) OptionalThrows ;} \)  \([\text{klabel('AbstractMethodDec')}]\)

**SYNTAX**  \( \text{AbstractMethodMod ::= Public} \)

\[ | \text{Abstract} \]

**SYNTAX**  \( \text{AnnoAbstractMethodMod ::= Anno} \)

\[ | \text{AbstractMethodMod} \]

### Annotations

**SYNTAX**  \( \text{Anno ::=@ TypeName ( ElemValPairList )} \)  \([\text{klabel('Anno')}]\)

\[ | @ \text{TypeName ( ElemVal )} \)  \([\text{klabel('SingleElemAnno')}]\)

\[ | @ \text{TypeName} \)  \([\text{klabel('MarkerAnno')}]\)

**SYNTAX**  \( \text{ElemVal ::= Exp} \)

\[ | \text{Anno} \]

\[ | \{ \text{ElemValList} \} \)  \([\text{klabel('ElemValArrayInit')}]\)

\[ | \{ \text{ElemValList ,} \} \)  \([\text{klabel('ElemValArrayInit')}]\)

**SYNTAX**  \( \text{ElemValPair ::= Id = ElemVal} \)  \([\text{klabel('ElemValPair')}]\)

### AnnotationTypes

**SYNTAX**  \( \text{AnnoDecHead ::= AnnoInterfaceModList @ interface Id} \)  \([\text{klabel('AnnoDecHead')}]\)

**SYNTAX**  \( \text{AnnoDec ::= AnnoDecHead \{ AnnoElemDecList \}} \)  \([\text{klabel('AnnoDec')}]\)

**SYNTAX**  \( \text{AnnoElemDec ::= ConstantDec} \)

\[ | \text{ClassDec} \]

\[ | \text{InterfaceDec} \]

\[ | \text{EnumDec} \]

\[ | \text{AnnoDec} \]

\[ | \text{SemiColon} \]

338
Syntax  \( \text{AnnoElemDec} ::= \text{AbstractMethodModList} \ \text{Type} \ \text{Id}() \ \text{OptionalDefaultVal} \ ; \)

\[ \text{[klabel('AnnoMethodDec)\]} \]

Syntax  \( \text{DefaultValue} ::= \text{default} \ \text{ElemVal} \ [\text{klabel('DefaultValue)\]} \]

Constant Declarations

Syntax  \( \text{ConstantDec} ::= \text{AnnoConstantModList} \ \text{Type} \ \text{VarDecList} \ ; \) [klabel('ConstantDec)\]

Syntax  \( \text{ConstantMod} ::= \text{Public} \)

\begin{align*}
| & \text{Static} \\
| & \text{Final}
\end{align*}

Syntax  \( \text{AnnoConstantMod} ::= \text{Anno} \)

\begin{align*}
| & \text{ConstantMod}
\end{align*}

Interface Declarations

Syntax  \( \text{InterfaceDecHead} ::= \text{AnnoInterfaceModList} \ \text{interface} \ \text{Id} \ \text{OptionalTypeParams} \ \text{OptionalExtendsInterfaces} \)

\[ \text{[klabel('InterfaceDecHead)\]} \]

Syntax  \( \text{InterfaceDec} ::= \text{InterfaceDecHead} \ \{ \ \text{InterfaceMemberDecList} \ \} \) [klabel('InterfaceDec)\]

Syntax  \( \text{ExtendsInterfaces} ::= \text{extends} \ \text{InterfaceTypeList} \) [klabel('ExtendsInterfaces)\]

Syntax  \( \text{InterfaceMemberDec} ::= \text{ConstantDec} \)

\begin{align*}
| & \text{AbstractMethodDec} \\
| & \text{ClassDec} \\
| & \text{InterfaceDec} \\
| & \text{SemiColon}
\end{align*}

Syntax  \( \text{InterfaceMod} ::= \text{Public} \)

\begin{align*}
| & \text{Private} \\
| & \text{Protected} \\
| & \text{Abstract} \\
| & \text{Final} \\
| & \text{Static} \\
| & \text{StrictFP}
\end{align*}

Syntax  \( \text{AnnoInterfaceMod} ::= \text{Anno} \)

\begin{align*}
| & \text{InterfaceMod}
\end{align*}
D.6 Module PACKAGE-SYNTAX

**SYNTAX**  
CompilationUnit ::= OptionalPackageDec ImportDecList TypeDecList  [klabel('CompilationUnit)]

**SYNTAX**  
TypeDec ::= ClassDec  
| InterfaceDec  
| SemiColon

**SYNTAX**  
ImportDec ::= import TypeName ;  [klabel('TypeImportDec)]  
| import PackageName . * ;  [klabel('TypeImportOnDemandDec)]  
| import static TypeName . Id ;  [klabel('StaticImportDec)]  
| import static TypeName . * ;  [klabel('StaticImportOnDemandDec)]

**SYNTAX**  
PackageDec ::= AnnoList package PackageName ;  [klabel('PackageDec)]

D.7 Module NAME-SYNTAX

**SYNTAX**  
PackageName ::= IdList  [klabel('PackageName)]

**SYNTAX**  
AmbName ::= Id  [klabel('AmbName)]  
| AmbName . Id  [klabel('AmbName)]

**SYNTAX**  
PackageOrTypeName ::= Id  [klabel('PackageOrTypeName)]  
| PackageOrTypeName . Id  [strict(1), klabel('PackageOrTypeName)]

**SYNTAX**  
ExprName ::= Id  [klabel('ExprName)]  
| AmbName . Id  [klabel('ExprName)]

**SYNTAX**  
TypeName ::= Id  [klabel('TypeName)]  
| PackageOrTypeName . Id  [strict(1), klabel('TypeName)]

**SYNTAX**  
KLabel ::= 'MethodName

**SYNTAX**  
MethodName ::= Id  
| AmbName . Id

D.8 Module LIST-SYNTAX

**SYNTAX**  
InterfaceTypeList ::= List{InterfaceType ,", "}  [klabel('InterfaceTypeList)]

**SYNTAX**  
ExceptionTypeList ::= List{ExceptionType ,", "}  [klabel('ExceptionTypeList)]
SYNTAX  \[ \text{IdList} ::= \text{List}\{\text{Id} , "\cdot"\} \]  [klabel('IdList')]

SYNTAX  \[ \text{TypeDecList} ::= \text{List}\{\text{TypeDec} , "\cdot"\} \]  [klabel('TypeDecList')]

SYNTAX  \[ \text{VarDecList} ::= \text{List}\{\text{VarDec} , "\cdot"\} \]  [klabel('VarDecList')]

SYNTAX  \[ \text{ImportDecList} ::= \text{List}\{\text{ImportDec} , "\cdot"\} \]  [klabel('ImportDecList')]

SYNTAX  \[ \text{ActualTypeArgList} ::= \text{List}\{\text{ActualTypeArg} , "\cdot"\} \]  [klabel('ActualTypeArgList')]

SYNTAX  \[ \text{ClassOrInterfaceTypeList} ::= \text{List}\{\text{ClassOrInterfaceType} , "\cdot"\&\} \]  [klabel('ClassOrInterfaceTypeList')]

SYNTAX  \[ \text{TypeParamList} ::= \text{List}\{\text{TypeParam} , "\cdot"\} \]  [klabel('TypeParamList')]

SYNTAX  \[ \text{AbstractMethodModList} ::= \text{List}\{\text{AbstractMethodMod} , "\cdot"\} \]  [klabel('AbstractMethodModList')]

SYNTAX  \[ \text{AnnoAbstractMethodModList} ::= \text{List}\{\text{AnnoAbstractMethodMod} , "\cdot"\} \]  [klabel('AnnoAbstractMethodModList')]

SYNTAX  \[ \text{AnnoMethodModList} ::= \text{List}\{\text{AnnoMethodMod} , "\cdot"\} \]  [klabel('AnnoMethodModList')]

SYNTAX  \[ \text{AnnoVarModList} ::= \text{List}\{\text{AnnoVarMod} , "\cdot"\} \]  [klabel('AnnoVarModList')]

SYNTAX  \[ \text{AnnoClassModList} ::= \text{List}\{\text{AnnoClassMod} , "\cdot"\} \]  [klabel('AnnoClassModList')]

SYNTAX  \[ \text{AnnoConstrModList} ::= \text{List}\{\text{AnnoConstrMod} , "\cdot"\} \]  [klabel('AnnoConstrModList')]

SYNTAX  \[ \text{AnnoConstantModList} ::= \text{List}\{\text{AnnoConstantMod} , "\cdot"\} \]  [klabel('AnnoConstantModList')]

SYNTAX  \[ \text{AnnoFieldModList} ::= \text{List}\{\text{AnnoFieldMod} , "\cdot"\} \]  [klabel('AnnoFieldModList')]

SYNTAX  \[ \text{AnnoInterfaceModList} ::= \text{List}\{\text{AnnoInterfaceMod} , "\cdot"\} \]  [klabel('AnnoInterfaceModList')]

SYNTAX  \[ \text{AnnoList} ::= \text{List}\{\text{Anno} , "\cdot"\} \]  [klabel('AnnoList')]

SYNTAX  \[ \text{AnnoElemDecList} ::= \text{List}\{\text{AnnoElemDec} , "\cdot"\} \]  [klabel('AnnoElemDecList')]

SYNTAX  \[ \text{InterfaceMemberDecList} ::= \text{List}\{\text{InterfaceMemberDec} , "\cdot"\} \]  [klabel('InterfaceMemberDecList')]

SYNTAX  \[ \text{ElemValPairList} ::= \text{List}\{\text{ElemValPair} , "\cdot"\} \]  [klabel('ElemValPairList')]

341
**SYNTAX**  \(ElemValList ::= List\{ElemVal, "\}" [klabel('ElemValList)]

**SYNTAX**  \(StringPartList ::= List\{StringPart, "\}" [klabel('StringPartList)]

**SYNTAX**  \(EnumConstList ::= List\{EnumConst, "\}" [klabel('EnumConstList)]

**SYNTAX**  \(ClassBodyDecList ::= List\{ClassBodyDec, "\}" [klabel('ClassBodyDecList)]

**SYNTAX**  \(BlockStmList ::= List\{BlockStmt, "\}" [klabel('BlockStmList)]

**SYNTAX**  \(SwitchGroupList ::= List\{SwitchGroup, "\}" [klabel('SwitchGroupList)]

**SYNTAX**  \(SwitchLabelList ::= List\{SwitchLabel, "\}" [klabel('SwitchLabelList)]

**SYNTAX**  \(Exps ::= List\{Exp, "\}" [seqstrict]

**SYNTAX**  \(Dims ::= List\{Dim, "\}"

**SYNTAX**  \(DimExps ::= List\{DimExp, "\}" [strict]

**SYNTAX**  \(VarInits ::= List\{VarInit, "\}"

**SYNTAX**  \(CatchClauses ::= List\{CatchClause, "\}" [strict]

**SYNTAX**  \(Params ::= List\{Param, "\}"
D.9  Module LEXICAL-SYNTAX

Identifiers

```
SYNTAX  Id ::= ID  [klabel('Id)]
```

```
SYNTAX  ID ::= Token{[A−Za−z_][A−Za−z0−9_]*}  [onlyLabel]
```

LineTerminators

```
SYNTAX  LineTerminator ::= Token{[\r]}  [onlyLabel]
  | Token{[\n]}  [onlyLabel]
  | Token{[\r][\n]}  [onlyLabel]
```
Modifiers

\begin{verbatim}
SYNTAX  Public ::= public [klabel('Public)]
SYNTAX  Private ::= private [klabel('Private)]
SYNTAX  Protected ::= protected [klabel('Protected)]
SYNTAX  Abstract ::= abstract [klabel('Abstract)]
SYNTAX  Final ::= final [klabel('Final)]
SYNTAX  Static ::= static [klabel('Static)]
SYNTAX  Native ::= native [klabel('Native)]
SYNTAX  Transient ::= transient [klabel('Transient)]
SYNTAX  Volatile ::= volatile [klabel('Volatile)]
SYNTAX  StrictFP ::= strictfp [klabel('StrictFP)]
SYNTAX  Synchronized ::= synchronized [klabel('Synchronized)]
\end{verbatim}

D.10 Module LITERAL-SYNTAX

BooleanLiterals

\begin{verbatim}
SYNTAX  BoolLiteral ::= Boolean [klabel('Bool)]
SYNTAX  Boolean ::= true [onlyLabel, klabel('True)]
SYNTAX  Boolean ::= false [onlyLabel, klabel('False)]
\end{verbatim}

CharacterLiterals

\begin{verbatim}
SYNTAX  CharContent ::= SingleChar [klabel('Single)]
        | UnicodeEscape
        | EscapeSeq
\end{verbatim}
EscapeSequences

**SYNTAX**  
\[\text{EscapeSeq} ::= \text{OctaEscape} \mid \text{NamedEscape}\]

**SYNTAX**  
\[\text{NamedEscape} ::= \text{Token}\{\"[bntfr\\]\\}\} \text{[onlyLabel, klabel('NamedEscape')]}
\]

**SYNTAX**  
\[\text{OctaEscape} ::= \text{Token}\{\"[0 \sim 7]\\}\} \text{[onlyLabel, klabel('OctaEscape1')]}
\]
\[\mid \text{Token}\{\"[0 \sim 3][0 \sim 7]\\}\} \text{[onlyLabel, klabel('OctaEscape2')]}
\]
\[\mid \text{Token}\{\"[4 \sim 7][0 \sim 7]\\}\} \text{[onlyLabel, klabel('OctaEscape2')]}
\]
\[\mid \text{Token}\{\"[0 \sim 3][0 \sim 7][0 \sim 7]\\}\} \text{[onlyLabel, klabel('OctaEscape3')]}
\]

FloatingPointLiterals

**SYNTAX**  
\[\text{FloatLiteral} ::= \text{Float} \text{[klabel('Float')]}
\]

IntegerLiterals

**SYNTAX**  
\[\text{DeciLiteral} ::= \text{Token}\{[1 \sim 9][0 \sim 9] \ast [IL]\\}\} \text{[onlyLabel]}
\]

**SYNTAX**  
\[\text{HexaLiteral} ::= \text{Token}\{[0][xX][0 \sim 9a \sim f][A \sim F]\} + [IL]\\} \text{[onlyLabel]}
\]

**SYNTAX**  
\[\text{OctaLiteral} ::= \text{Token}\{[0][0 \sim 7] + [IL]\\} \text{[onlyLabel]}
\]

**SYNTAX**  
\[\text{IntLiteral} ::= \text{DeciLiteral} \text{[klabel('Deci')]}
\]
\[\mid \text{HexaLiteral} \text{[klabel('Hexa')]}
\]
\[\mid \text{OctaLiteral} \text{[klabel('Octa')]}
\]

NullLiteral

**SYNTAX**  
\[\text{NullLiteral} ::= \text{null} \text{[klabel('Null')]}
\]

StringLiterals

**SYNTAX**  
\[\text{StringLiteral} ::= '' \text{StringPartList} '' \text{[klabel('String')]}
\]

**SYNTAX**  
\[\text{StringPart} ::= \text{StringChars} \text{[klabel('Chars')]}
\]
\[\mid \text{UnicodeEscape}
\]
\[\mid \text{EscapeSeq}
\]

**SYNTAX**  
\[\text{UnicodeEscape} ::= \text{Token}\{\\\text{\\}[u] +[0 \sim 9a \sim f A \sim F][0 \sim 9a \sim f A \sim F][0 \sim 9a \sim f A \sim F]\\}\} \text{[onlyLabel, klabel('UnicodeEscape')]}
\]

345
D.11 Module OPTIONAL-SYNTAX

```
SYNTAX  None ::= [onlyLabel, klabel('None)]

SYNTAX  OptionalId ::= Id  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalExp ::= Exp  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalWildcardBound ::= WildcardBound  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalTypeArgs ::= TypeArgs  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalTypeParams ::= TypeParams  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalTypeBound ::= TypeBound  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalThrows ::= Throws  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalDefaultVal ::= DefaultVal  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalExtendsInterfaces ::= ExtendsInterfaces  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalInterfaces ::= Interfaces  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalPackageDec ::= PackageDec  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalSuper ::= Super  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalConstrInv ::= ConstrInv  [prefer, klabel('Some)]
            |  None

SYNTAX  OptionalClassBody ::= ClassBody  [prefer, klabel('Some)]
            |  None
```
SYNTAX  OptionalEnumBodyDecs ::= EnumBodyDecs [prefer, klabel('Some)]
|  None

SYNTAX  OptionalEnumConstArgs ::= EnumConstArgs [prefer, klabel('Some)]
|  None
References


978-1-4503-3300-9. DOI: 10.1145/2676726.2676982 URL: http://doi.acm.org/10.1145/2676726.2676982


FSL. K Framework. URL: http://kframework.org

FSL. MatchC project. URL: http://fsl.cs.illinois.edu/index.php/Special:MatchCOnline

Brian Goetz. Translation of Lambda Expressions. URL: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html


350


[37] *OpenJDK project*. url: [http://openjdk.java.net/](http://openjdk.java.net/)


