ActionScript® 4.0 Language Specification

Avik Chaudhuri, Bernd Mathiske, Krzysztof Palacz, and Basil Hosmer

Adobe Systems
AS4@adobe.com

December 13, 2012

ActionScript® 4.0 Language Specification

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Change Log

Oct 30 2012 Draft released to CAB.

Nov 1 2012 Fixed bug around syntactic ordering of access controls and attributes on function definitions and class definitions: cf. occurrences of the new nonterminal *Modifier* in the grammar (Chapter 3) and in the pruning rules (Chapter 4).

Nov 30 2012 Fixed coercion semantics of `as` to perform some "bitcast" numeric conversions instead of throwing range errors: cf. occurrences of "coercion operator" in type inference (Chapter 7) and the rules for $T' :: op(e\ \texttt{coerce}\ T)$ (Chapter 8).

Dec 5 2012 Added syntactic support for variable-length unicode escape sequences: cf. occurrences of the new nonterminal VariableLengthUnicodeEscapeSequence in the grammar (see Chapter 3).

Dec 5 2012 Extended the syntax of type expressions to include array and function types: cf. the nonterminal *TypeExpression* in the grammar (Chapter 3).

Dec 5 2012 Fixed bugs in the canonicalization of compound assignments and prefix/postfix operations: cf. rules for deriving canonical forms (Chapter 5).

Dec 5 2012 Removed the `for-in` construct: cf. the nonterminal *ForStatement* in the grammar (Chapter 3) and the rules for deriving canonical forms (Chapter 5).

Dec 12 2012 Added singleton types for numeric literals, which are now used to type numeric literals; added a notion of numeric literals fitting in types to describe the earlier typing scheme for numeric literals, and used it to extend the notion of promotion and the definition of union; tweaked the rules for typing arithmetic operations to perform expected promotions of numeric literals (Chapter 7).

Dec 13 2012 Fixed typing rules for shifting, which earlier took into account the type of the shift amount in determining the type of the result of a shift, instead of simply preserving the type of the value to be shifted: the type of the shift amount is now `int` (Chapter 7).

Dec 13 2012 Removed canonicalization rules for conditional expressions: constant evaluation was not possible earlier due to canonicalization (Chapter 5); fixed typing rules for conditional expressions (Chapter 7).

Contents

Part I

Overview

Chapter 1

Syntax and Semantics

This chapter provides an overview of the syntax and semantics of ActionScript 4.0 (AS4).

1.1 Syntactic Model

1.1(1) The syntax of the language defines the interpretation of a sequence of characters (the text of a program) as a syntax tree that represents a syntactically valid program in the language. This interpretation involves the following steps:

1. The processes of *scanning* (Section 3.2) and *parsing* (Section 3.3) translate a sequence of characters (the text of a program) to an intermediate syntax tree.
2. The processes of *unit configuration* (Section 4.1) and *enforcement of syntactic restrictions* (Section 4.2) eliminate parts of the syntax tree, and discards the pruned intermediate syntax tree unless it satisfies various conditions of syntactic validity.

1.1(2) A syntactically valid program consists of several units which contain type definitions.

1.1(3) An IDE for the language may enforce further restrictions on the syntax tree of a valid program in the language, the details of which are not considered further in the specification.

1.1(4) To simplify the presentation of semantics in the remainder of this specification, we assume that some forms of syntax trees are encoded away in terms of other canonical forms (Section 5.1). An implementation does not need to perform such encodings; the semantic rules for non-canonical forms can be derived through those for their corresponding encodings.

1.2 Compilation Model

1.2(1) An *executable* represents some type definitions and the name of a distinguished static method of some class among those classes, that serves as the *entry point*. It is assumed that the code in an executable has undergone (at least) the processes of resolution (Chapter 6) and inference (Chapter 7).

1.2(2) A *library* is an executable whose entry point is ignored.

1.2(3) A program is *compiled* to an executable, against a set of libraries.

1.2(4) There exists a pre-compiled library, known as the *standard library*, that contains definitions of built-in classes and interfaces, and a trivial entry point. The built-in classes include at least the classes corresponding to value types `int`, `uint`, `long`, `ulong`, `double`, `float`, `byte`, `bool`, the type `String` that has a method `concat(string:String)`, the types of reified type objects `Class` and `Interface` that can at least reflect on the type definitions and the definitions of their members (and in particular, reflect on metadata associated with such type definitions and the definitions of their members), the generic types `Array<T>` (which is a desugaring of `[]` T) and `ArrayList<T>` for every type T, the type of regular expressions with syntactic support `RegExp` that has a constructor `RegExp(pattern:String,flags:String)`, and the type of all objects `Object` that has methods `equal(object:Object):bool`, `identical(object:Object):bool`, and `toString():String`. Their remaining details are not considered further in the specification.

1.2(5) Compilation proceeds as follows:

1. *Compile-time information* is derived for the classes and interfaces that appear in the libraries and the program (Chapter 6).
2. The expressions and statements in the program are *statically verified* by inferring types and compile-time constants, yielding an executable (Chapter 7).
3. Compilation fails if any of the above steps reports a compile-time error.
4. Otherwise the executable is returned.

1.3 Execution Model

1.3(1) An executable is executed to produce a sequence of observables, against a run-time representation of types, which is initially empty and is grown by executing prior executables, including the standard library.

1.3(2) Execution of an executable proceeds as follows:

1. The static method corresponding to the entry point of the executable is executed, which requires the *initialization* of the class that contains that static method, which may in turn trigger the initialization of other classes and interfaces, thereby building run-time representations of these types (Chapter 8).
2. As part of initialization of types, other methods may be run, which involves the evaluation of statements and expressions therein, possibly yielding observables and other unobservable values (Chapter 9).

1.3(3) A class must be *linked* before it is initialized. Linking a class regenerates compile-time information on that class.

1.3(4) A method must be *dynamically verified* before it is run. Dynamically verifying a method prior to running it enforces the same constraints as would be by static verification of that method.

1.3(5) A type must be linked before it participates in verification.

1.3(6) Conceptually, if a program has been compiled against a library, then executables that define the types (classes and interfaces) in that library (which presumably the program relies on) should be executed prior to executing the executable that the program compiles to so that those types already have run-time representations. In practice, this requirement can be slightly relaxed: a type must be initialized prior to its use at run time. This requirement is enforced by regenerating compile-time information at link time and performing dynamic verification of code prior to execution.

Chapter 2

Implementation Notes

This chapter outlines the criteria for implementation correctness and points out some degrees of freedom in implementing this specification.

2.1 Correctness

2.1(1) A runtime for the language is considered correct for an executable when the executable produces an observable if and only if this specification predicts that it should produce the same observable (modulo system constraints).

2.1(2) A compiler for the language is considered correct for a program when it returns an executable if and only if this specification predicts that it should return some executable, and these executables when run by a correct runtime produce the same observable (modulo system constraints).

2.2 Intermediate Representations

2.2(1) An executable that is returned by compilation can be encoded in an intermediate representation format, such as the ActionScript Byte Code format, for mobility, and decoded from that format upon loading for execution without any loss of information. The details of such formats are not considered further in this specification.

2.2(2) The compiler and the runtime share knowledge of a set of intrinsic operations that manipulate intrinsic data structures (Chapter 8). These intrinsics serve only to specify a semantic model for the language; they should otherwise be considered abstract, and can be replaced by efficient implementations that preserve the same semantic behaviors of their abstract specifications, i.e., are correct.

2.2(3) Some of the intrinsic operations may be available as language APIs (with possibly native implementations), whereas others may correspond to "bytecode instructions." The details of such implementation strategies are not considered further in this specification.

2.2(4) In this specification, expressions are translated to intrinsic operations as part of dynamic verification. An implementation may choose to perform this translation at compile time instead, as part of generating executable code: in particular, such an implementation could generate bytecode instructions and language APIs corresponding to those intrinsic operations. In such an implementation, dynamic verification would have to be performed on the instrinsic operations instead. The details of such implementation strategies are not considered further in this specification.

2.2(5) An executable, along with the libraries used by the compiler to produce it, could be further compiled "ahead-of-time" to native code and executed as such, without further linking and verification. The details of such execution strategies are not considered further in this specification, except to note that the semantic behavior of such native code should be the same as what is predicted by this specification, when the same libraries are loaded prior to executing the same executable.

Part II

Syntax

Chapter 3

Scanning and Parsing

This chapter specifies how a sequence of characters is interpreted as a syntax tree by the processes of scanning and parsing, to obtain a program that may not yet be syntactically valid. The next chapter specifies further rules for pruning such a syntax tree, by the processes of unit configuration and enforcement of syntactic restrictions, to obtain a syntactically valid program.

3.1 Preliminaries

3.1.1 Grammars

3.1(1) A grammar is specified by a set of *rules*. A rule defines a *nonterminal* by a set of *productions*. A production is a sequence of *terminals* and nonterminals, possibly with some side conditions.

3.1(2) A grammar identifies the sequences of terminals that *match* a nonterminal. A sequence of terminals A matches a nonterminal B if there is a production in the rule for the nonterminal B in the grammar that, upon substituting every nonterminal in that production with some sequence of terminals that matches it, becomes the sequence of terminals A. Furthermore, side conditions may appear in various positions in a production, and the conditions must be satisfied at those positions. In particular, side conditions may disambiguate ambiguous matches or restrict possible matches, based on context.

3.1(3) A *syntax tree* is an ordered tree that represents how a sequence of terminals match a nonterminal. The terminals are the leaves of the tree, and the nonterminal is the root. Furthermore, intermediate nonterminals are the internal nodes of the tree. Any subtree is a syntax tree that represents how the subsequence of terminals that are leaves of that subtree match the intermediate nonterminal that is the root of that subtree. The children of any parent are the nonterminals and terminals that appear in some production of the parent, and they are ordered by the order in which they appear in that production from left to right.

3.1(4) A syntax tree A is *nested* by another syntax tree B if A is a (proper) subtree of B. A is nested by B *without crossing* a syntax tree C if A is nested by B but either A is not nested by C or C is not nested by B. In particular, if A is nested by B, then it follows that A is not B, A is nested by B without crossing A, and A is nested by B without crossing B.

3.1(5) An *ordered traversal* of a syntax tree is a traversal of the nodes of the tree in which a parent is visited before its children, and the children of a parent are visited in order. A node A appears *earlier* or *later* than another node B in the syntax tree if a ordered traversal visits A before or after B, respectively. By extension, a subtree A appears earlier or later than another subtree B if the root of A appears earlier or later than the root of B, respectively.

3.1.2 Programs

3.1(6) The syntax of the language is specified by a *syntactic grammar*, which in turn relies on a *lexical grammar*. The nonterminals and terminals of the syntactic grammar are *syntactic nonterminals* and *syntactic terminals*, respectively. The nonterminals and terminals of the lexical grammar are *lexical nonterminals* and *lexical terminals*, respectively.

3.1(7) A lexical terminal is a sequence of Unicode code units. A sequence of lexical terminals that matches the lexical nonterminal *InputElementOperand* or *InputElementOperator* is an *input element*. An input element that is a syntactic terminal is a *token*. Any other input element is a *token separator*.

3.1(8) There are two distinct *parser contexts* defined by the lexical nonterminals InputElementOperator and InputElementOperand. (These parser contexts are required to disambiguate / as the prefix of a *RegularExpressionLiteral* or as a binary operation.) In a particular parser context, input elements (tokens and token separators) must match the particular lexical nonterminal that defines that parser context. The parser switches into a particular parser context before or after it matches particular nonterminals (i.e., when the current position immediately precedes or immediately succeeds some input text that matches particular nonterminals), as described below.

1. The parser is initially in the parser context defined by InputElementOperator, and switches into that parser context after matching a *PrimaryExpression*.
2. The parser switches into the parser context defined by InputElementOperand before matching a *PrimaryExpression*.

3.1(9) The input elements that serve as token separators are Whitespace, LineTerminator, and Comment. By separating tokens, they provide flexibility in how the text of a program is formatted. Token separators are discarded from the output of scanning (which then becomes the input of parsing).

3.1(10) *Scanning* is the process of matching some text (a sequence of lexical terminals) to a sequence of tokens, some of which may be separated by token separators. The tokens must be maximal, in the following sense: if both A, B, and $A\,B$ are tokens, then the text $A\,B$ is scanned as the token $A\,B$, instead of the token A followed by the token B.

3.1(11) *Parsing* is the process of matching a sequence of tokens to a syntactic nonterminal (satisfying any associated side conditions).

3.1(12) A *syntactically valid program* is a sequence of lexical terminals (the text of the program) that, upon scanning, can be parsed to the syntactic nonterminal *Program* without any remaining text.

3.1.3 Rules, Productions, Terminals, and Nonterminals

3.1(13) A rule spans several lines; the first line contains the nonterminal that is defined by the rule, and each remaining line contains a production for that nonterminal. Rules are separated by blank lines.

3.1(14) A production is a sequence of terminal and nonterminal symbols with optional side conditions at various positions in the sequence.

3.1(15) Names of syntactic nonterminals begin with uppercase letters and are in slanted sans serif font, e.g., *Expression*. Names of lexical nonterminals (which may also be syntactic terminals) begin with uppercase letters and are in sans serif font, e.g. NumericLiteral. Lexical terminals (which may also be syntactic terminals) represent sequences of Unicode code units that are either represented by literal characters in typewriter font, e.g. `{` or `function`, or described by Unicode categories.

3.1(16) Identifiers that are represented in typewriter font have special meaning in the context in which they occur in the grammar. Such identifiers may or may not be globally reserved. Globally reserved identifiers are listed in the lexical nonterminal Keyword.

3.1.4 Side Conditions

Side conditions rely on the following notation. ($\mathcal{X}$ is a metavariable denoting some grammatical entity).

3.1(17) Literal non-blank characters in a typewriter font are taken from the ISO Latin-1 character set and represent the corresponding Unicode code units.

3.1(18) ϵ is matched by the empty sequence.

3.1(19) $\mathcal{X}_{\texttt{opt}}$ is matched by either the empty sequence or a sequence that matches $\mathcal{X}$.

3.1(20) `U+` followed by four HexadecimalDigits (hexadecimal digits) is standard notation for a Unicode code unit.

3.1(21) ⟨lookahead not $\mathcal{X}$⟩ requires that any following nonterminal is not matched by a sequence of Unicode code units that matches $\mathcal{X}$

3.1(22) ⟨but not $\mathcal{X}$⟩ requires that the preceding nonterminal is not matched by a sequence of Unicode code units that matches $\mathcal{X}$.

3.1(23) ⟨any Unicode $\mathcal{X}$⟩ is any Unicode code unit denoted by $\mathcal{X}$.

3.1(24) … or … means choice.

3.2 Lexical Grammar

3.2.1 Input Elements

InputElementOperand [1]

1 Whitespace
2 LineTerminator
3 Comment
4 IdentifierOrKeyword
5 NumericLiteral
6 StringLiteral
7 Punctuator ⟨but not `/` or `/=`⟩
8 RegularExpressionLiteral

[1] AS3 has four parsing contexts, and correspondingly, four lexical nonterminals to recognize input elements in those parsing contexts. In contrast, AS4 has just two. This simplification is possible because AS4 does not syntactically support E4X XML literals (but continues to support ECMAScript regular-expression literals).

InputElementOperator

9 Whitespace
10 LineTerminator
11 Comment
12 IdentifierOrKeyword
13 NumericLiteral
14 StringLiteral
15 Punctuator

3.2.2 Whitespace and Line Terminators

Whitespace [2]

16 U+0009
17 U+000B
18 U+000C
19 U+FEFF
20 ⟨any Unicode Zs⟩

2 Any Unicode Cf can be used within comments and strings. Outside of comments and strings, the following three Unicode code units have the given meanings:

- U+200C → IdentifierPart
- U+200D → IdentifierPart
- U+FEFF → Whitespace

LineTerminator

21 U+000A
22 U+000D
23 U+2028
24 U+2029
25 U+000D U+000A

3.2.3 Comments

Comment

26 MultiLineComment
27 SingleLineComment

MultiLineComment

28 /* MultiLineCommentCharacters$_{opt}$ */

MultiLineCommentCharacters

29 SourceCharacter ⟨but not *⟩ MultiLineCommentCharacters$_{opt}$
30 * ⟨lookahead not /⟩ MultiLineCommentCharacters$_{opt}$

SingleLineComment

31 // SingleLineCommentCharacters$_{opt}$

SingleLineCommentCharacters

32 SourceCharacter ⟨but not LineTerminator⟩ SingleLineCommentCharacters$_{opt}$

SourceCharacter

33 ⟨any Unicode code unit⟩

3.2.4 Identifiers

Identifier

34 IdentifierOrKeyword ⟨but not Keyword⟩

IdentifierOrKeyword [3]

35 IdentifierStart

36 IdentifierOrKeyword IdentifierPart

[3] Unicode escape sequences may be used to spell the names of identifiers that would otherwise be keywords. This is in contrast to ECMAScript.

IdentifierStart

37 UnicodeLetter

38 $

39 _

40 \ fixedLengthUnicodeEscapeSequence

IdentifierPart

41 IdentifierStart

42 UnicodeCombiningMark

43 UnicodeDigit

44 UnicodeConnectorPunctuation

45 U+200C ⟨ZWNJ⟩

46 U+200D ⟨ZWJ⟩

UnicodeLetter

47 ⟨any Unicode Lu or Ll or Lt or Lm or Lo or Nl⟩

UnicodeCombiningMark

48 ⟨any Unicode Mn or Mc⟩

UnicodeDigit

49 ⟨any Unicode Nd⟩

UnicodeConnectorPunctuation

50 ⟨any Unicode Pc⟩

3.2.5 Keywords and Punctuators

Keyword [4]

51 `as`

52 `break`

53 `case`

54 `catch`

55 `class`

56 `continue`

57 `default`

58 `defer`

59 `do`

60 `else`

61 `false`

62 `finally`

63 `for`

64 `function`

```
if
import
interface
internal
is
let
new
null
package
private
protected
public
return
super
switch
this
throw
true
try
var
void
while
```

4 Keywords are reserved words that have special meanings. Some Identifiers have special meanings in some syntactic contexts, but are not Keywords; such Identifiers are contextually reserved.

The following AS3 keywords are no longer in AS4: `delete`, `include`, `instanceof`, `namespace`, `typeof`, `use`, `with`, `in`. The `const` keyword is replaced by `let`. The keyword `defer` is introduced in AS4.

Punctuator [5]

```
.
!
!=
!==
%
%=
&
&=
&&
&&=
*
*=
+
+=
++
-
-=
--
=
==
===
>
>=
>>
```

111 >>=
112 ^
113 ^=
114 |
115 |=
116 ||
117 ||=
118 :
119 (
120)
121 [
122]
123 {
124 }
125 ~
126 ,
127 ;
128 ?
129 @
130 /
131 /=
132 <
133 <=
134 <<
135 <<=
136 =>

5 The following AS3 punctuators are no longer in AS4: `.<`, `..`, `::`, `>>>`, `>>>=`, `...`. The punctuator `@` is repurposed. The punctuator `=>` is introduced.

3.2.6 Numeric Literals

NumericLiteral [6]

137 DecimalLiteral
138 HexadecimalIntegerLiteral

6 The source character immediately following a NumericLiteral may be an IdentifierStart. This is in contrast to ECMAScript. This might be useful to distinguish literals for unsigned numbers, floating point numbers, and so on, in the future.

DecimalLiteral

139 DecimalDigits . DecimalDigits$_{opt}$ ExponentPart$_{opt}$
140 . DecimalDigits ExponentPart$_{opt}$
141 DecimalDigits ExponentPart$_{opt}$

DecimalDigits

142 DecimalDigit DecimalDigits$_{opt}$

DecimalDigit

143 0
144 1
145 2
146 3

147 4
148 5
149 6
150 7
151 8
152 9

ExponentPart

153 ExponentIndicator Sign$_{\texttt{opt}}$ DecimalDigits

ExponentIndicator

154 e
155 E

Sign

156 +
157 -

HexadecimalIntegerLiteral

158 0x HexadecimalDigits
159 0X HexadecimalDigits

HexadecimalDigits

160 HexadecimalDigit HexadecimalDigits$_{\texttt{opt}}$

HexadecimalDigit

161 0
162 1
163 2
164 3
165 4
166 5
167 6
168 7
169 8
170 9
171 a
172 b
173 c
174 d
175 e
176 f
177 A
178 B
179 C
180 D
181 E
182 F

3.2.7 String Literals

StringLiteral

183 " DoubleStringCharacters$_{opt}$ "

184 ' SingleStringCharacters$_{opt}$ '

DoubleStringCharacters

185 DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters

186 SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter

187 SourceCharacter ⟨but not " or \ or LineTerminator⟩

188 \ EscapeSequence

189 LineContinuation

SingleStringCharacter

190 SourceCharacter ⟨but not ' or \ or LineTerminator⟩

191 \ EscapeSequence

192 LineContinuation

LineContinuation

193 \ LineTerminator

EscapeSequence [7]

194 CharacterEscapeSequence

195 0 ⟨lookahead not DecimalDigit⟩

196 HexadecimalEscapeSequence

197 FixedLengthUnicodeEscapeSequence

198 VariableLengthEscapeSequence

[7] During lexical analysis, a \EscapeSequence other than \VariableLengthEscapeSequence is translated to a single Unicode code unit, and a \VariableLengthEscapeSequence is translated to a single Unicode code point. This means that its interpretation does not affect the lexical structure (and therefore syntax) of the program. For example, `\n` is a string character that is interpreted as a line feed. This holds for UnicodeEscapeSequence as well, e.g., `\u000A`, in contrast to Java's treatment of Unicode escape sequences, which are interpreted before lexical analysis.

CharacterEscapeSequence

199 SingleEscapeCharacter

200 NonEscapeCharacter

SingleEscapeCharacter

201 '

202 "

203 \

204 b

205 f

206 n

207 r

208 t

209 v

NonEscapeCharacter

210 SourceCharacter ⟨but not EscapeCharacter or LineTerminator⟩

EscapeCharacter

211 SingleEscapeCharacter
212 DecimalDigit
213 `x`
214 `u`
215 `U`

HexadecimalEscapeSequence

216 `x` HexadecimalDigit HexadecimalDigit

FixedLengthUnicodeEscapeSequence

217 `u` HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit

VariableLengthUnicodeEscapeSequence

218 `U` `{` $\text{HexadecimalDigits}_{\texttt{opt}}$ `}`

3.2.8 Regular Expression Literals

RegularExpressionLiteral [8]

219 `/` RegularExpressionBody `/` $\text{RegularExpressionFlags}_{\texttt{opt}}$

[8] A RegularExpressionBody is never ϵ; instead of representing an empty regular expression, `//` starts a SingleLineComment. To specify an empty regular expression, use `/(?:)/`.

RegularExpressionBody

220 RegularExpressionFirstCharacter $\text{RegularExpressionCharacters}_{\texttt{opt}}$

RegularExpressionCharacters

221 RegularExpressionCharacter $\text{RegularExpressionCharacters}_{\texttt{opt}}$

RegularExpressionFirstCharacter

222 RegularExpressionNonTerminator ⟨but not `*` or `\` or `/` or `[`⟩
223 RegularExpressionBackslashSequence
224 RegularExpressionClass

RegularExpressionCharacter

225 RegularExpressionNonTerminator ⟨but not `\` or `/` or `[`⟩
226 RegularExpressionBackslashSequence
227 RegularExpressionClass

RegularExpressionBackslashSequence

228 `\` SourceCharacter

RegularExpressionNonTerminator

229 SourceCharacter ⟨but not LineTerminator⟩

RegularExpressionClass

230 `[` $\text{RegularExpressionClassCharacters}_{\texttt{opt}}$ `]`

RegularExpressionClassCharacters

231 RegularExpressionClassCharacter RegularExpressionClassCharacters$_{\texttt{opt}}$

RegularExpressionClassCharacter

232 RegularExpressionNonTerminator ⟨but not] or \⟩

233 RegularExpressionBackslashSequence

RegularExpressionFlags

234 IdentifierPart RegularExpressionFlags$_{\texttt{opt}}$

3.3 Syntactic Grammar

3.3.1 Types

TypeName

235 Identifier

236 *PackageName* . Identifier

TypedBinding

237 Identifier

238 Identifier : *Type*

Type

239 *

240 *StaticType*

StaticType[9]

241 *NominalType*

242 *ArrayType*

243 *FunctionType*

9 *ArrayType* and *FunctionType* are new forms of types in AS4.

NominalType

244 *TypeName*

245 *GenericType*

GenericType[10]

246 *TypeName* < *Types* >

10 In AS4, the *TypeName* must reference the built-in definition of `ArrayList` (which replaces AS3's `Vector`) or `Array` (which is new in AS4) and the *Types* must be a *Type*. Furthermore, the punctuator following the *TypeName* in AS4 is `<`, which replaces the non-traditional `.<` in AS3.

FunctionType

247 (*Types*$_{\texttt{opt}}$) => *Type*

Types

248 *Type*
249 *Type* , *Types*

ArrayType
250 `[ ]` *Type*

3.3.2 Primary Expressions

ArrayInitializer[11]
251 `new` *Dimension Type*
252 `new` *ArrayType* `{` *ArrayElements*$_{\mathtt{opt}}$ `}`

11 AS4 introduces new syntactic forms for array initializers. The forms of array initializers are restricted so that only single-dimensional array initializers can benefit from the special syntax. The introduction of multi-dimensional arrays in a future version will generalize this special syntax.

Dimension
253 `[` *Expression* `]`

ArrayElements
254 *Expression*
255 *Expression* , *ArrayElements*$_{\mathtt{opt}}$

FunctionInitializer
256 `function` Identifier$_{\mathtt{opt}}$ *FunctionBody*

FunctionSignature
257 `(` *Parameters*$_{\mathtt{opt}}$ `)` *ResultType*$_{\mathtt{opt}}$

Parameters[12]
258 *Parameter*
259 *OptionalParameters*
260 *Parameter* , *Parameters*

12 AS4, unlike AS3, does not support rest parameters.

OptionalParameters
261 *OptionalParameter*
262 *OptionalParameter* , *OptionalParameters*

OptionalParameter
263 *Parameter* `=` *Expression*

Parameter
264 *TypedBinding*

ResultType
265 `: void`
266 `:` *Type*

FunctionBody
267 *FunctionSignature Block*

PrimaryExpression [13]

268 `null`
269 `true`
270 `false`
271 `this`
272 NumericLiteral
273 StringLiteral
274 RegularExpressionLiteral
275 *ArrayInitializer*
276 *FunctionInitializer*

13 In AS3, primary expressions included *VectorInitializer*s, *XMLInitializer*s, and *XMLListInitializer*s; these are no longer supported in AS4. Furthermore, *ObjectInitializer*s are repurposed in AS4.

3.3.3 Expressions

ParenExpression

277 `(` *Expression* `)`

Arguments

278 `(` *ArgumentExpressions*$_{\texttt{opt}}$ `)`

ArgumentExpressions

279 *Expression*
280 *ArgumentExpressions* `,` *Expression*

MemberOperator

281 `.` Identifier

IndexOperator

282 `[` *Expression* `]`

SuperExpression

283 `super` *MemberOperator*

ReferenceExpression [14]

284 Identifier
285 *NominalType MemberOperator*
286 *BaseExpression MemberOperator*
287 *BaseExpression IndexOperator*

14 There is a parsing ambiguity between *PackageName* `.` Identifier `.` Identifier (produced by *NominalType MemberOperator*) and *ReferenceExpression*`.` Identifier `.` Identifier (produced by *BaseExpression MemberOperator*), when both *PackageName* and *ReferenceExpression* are Identifiers. This ambiguity is resolved by prefering the latter syntax tree during parsing, and if required, re-interpreting it as the former syntax tree later when knowledge of package names is available.

TypeExpression [15]

288 `:` *StaticType*

15 In AS3, a *StaticType* could appear as a stand-alone *Expression*, and would evaluate to a reified object corresponding to the type. In this model, static method calls on such a type were desugared to instance method calls on the reified object. In contrast, in AS4 static method calls on a type are distinguished from instance method calls on the reified object. Accordingly, *StaticType*s

cannot appear as stand-alone *Expression*s in AS4, instead a new syntactic form is introduced for *TypeExpression*s that evaluate to reified objects corresponding to types.

NewExpression

289 `new` *ObjectInitializer*

ObjectInitializer [16]

290 *NominalType Arguments*
291 *NominalType* ⟨lookahead not `(`⟩
292 *NominalType Arguments*$_{\texttt{opt}}$ `{` *FieldValuePairs* `}`

16 AS4 introduces a new syntactic form for *ObjectInitializer*s which can not only call constructors but also initialize fields.

FieldValuePairs

293 *FieldValuePair*
294 *FieldValuePair* `,` *FieldValuePairs*$_{\texttt{opt}}$

FieldValuePair

295 Identifier = *Expression*

BaseExpression

296 *PrimaryExpression*
297 *SuperExpression*
298 *NewExpression*
299 *ReferenceExpression*
300 *TypeExpression*
301 *ParenExpression*
302 *BaseExpression Arguments*

UnaryExpression [17]

303 *BaseExpression*
304 *BaseExpression* `is` *StaticType*
305 *BaseExpression* `as` *Type*
306 *ReferenceExpression* `++`
307 *ReferenceExpression* `--`
308 `++` *ReferenceExpression*
309 `--` *ReferenceExpression*
310 `+` *UnaryExpression*
311 `-` *UnaryExpression*
312 `~` *UnaryExpression*
313 `!` *UnaryExpression*

17 AS4 changes the operator precedence ordering for `is` and `as` to reduce unintentional errors. The semantics of `as` is also changed to mimic AS3's function call syntax for coercions, which is in turn removed.

MultiplicativeExpression

314 *UnaryExpression*
315 *MultiplicativeExpression* `*` *UnaryExpression*
316 *MultiplicativeExpression* `/` *UnaryExpression*
317 *MultiplicativeExpression* `%` *UnaryExpression*

AdditiveExpression

318 *MultiplicativeExpression*
319 *AdditiveExpression* + *MultiplicativeExpression*
320 *AdditiveExpression* - *MultiplicativeExpression*

ShiftExpression
321 *AdditiveExpression*
322 *ShiftExpression* << *AdditiveExpression*
323 *ShiftExpression* >> *AdditiveExpression*

RelationalExpression
324 *ShiftExpression*
325 *RelationalExpression* < *ShiftExpression*
326 *RelationalExpression* > *ShiftExpression*
327 *RelationalExpression* <= *ShiftExpression*
328 *RelationalExpression* >= *ShiftExpression*

EqualityExpression
329 *RelationalExpression*
330 *EqualityExpression* == *RelationalExpression*
331 *EqualityExpression* != *RelationalExpression*
332 *EqualityExpression* === *RelationalExpression*
333 *EqualityExpression* !== *RelationalExpression*

BitwiseANDExpression
334 *EqualityExpression*
335 *BitwiseANDExpression* & *EqualityExpression*

BitwiseXORExpression
336 *BitwiseANDExpression*
337 *BitwiseXORExpression* ^ *BitwiseANDExpression*

BitwiseORExpression
338 *BitwiseXORExpression*
339 *BitwiseORExpression* | *BitwiseXORExpression*

LogicalANDExpression
340 *BitwiseORExpression*
341 *LogicalANDExpression* && *BitwiseORExpression*

LogicalORExpression
342 *LogicalANDExpression*
343 *LogicalORExpression* || *LogicalANDExpression*

Expression
344 *LogicalORExpression*
345 *LogicalORExpression* ? *Expression* : *Expression*

3.3.4 Statements

Statement

346 *BreakStatement*
347 *ContinueStatement*
348 *DeferStatement*
349 *EmptyStatement*
350 *AssignmentStatement*
351 *ForStatement*
352 *IfStatement*
353 *LabeledStatement*
354 *BlockStatement*
355 *ReturnStatement*
356 *SuperStatement*
357 *SwitchStatement*
358 *ThrowStatement*
359 *TryStatement*
360 *WhileStatement*
361 *DoStatement*

BlockStatement [18]

362 *Block*

18 AS4 introduces block scoping, which replaces the non-traditional scoping rules of AS3 that involved hoisting. Accordingly, AS4 also introduces a new syntactic form for block statements.

Block

363 `{` *Directives*$_{\texttt{opt}}$ `}`

EmptyStatement

364 `;`

Assignment [19]

365 *Expression*
366 *ReferenceExpression* `=` *Expression*
367 *ReferenceExpression* `*=` *Expression*
368 *ReferenceExpression* `/=` *Expression*
369 *ReferenceExpression* `%=` *Expression*
370 *ReferenceExpression* `+=` *Expression*
371 *ReferenceExpression* `-=` *Expression*
372 *ReferenceExpression* `<<=` *Expression*
373 *ReferenceExpression* `>>=` *Expression*
374 *ReferenceExpression* `&=` *Expression*
375 *ReferenceExpression* `^=` *Expression*
376 *ReferenceExpression* `|=` *Expression*
377 *ReferenceExpression* `&&=` *Expression*
378 *ReferenceExpression* `||=` *Expression*

19 In AS3, *Assignment*s were *Expression*s. In AS4, they are restricted to *Statement*s to reduce unintentional errors and to plan for struct initializers and function calls with named parameter passing in the future.

Assignments

379 *Assignment*
380 *Assignment* `,` *Assignments*

AssignmentStatement

381 ⟨lookahead not { or function⟩ *Assignments* ;

LabeledStatement

382 Identifier : *Statement*

IfStatement

383 if *ParenExpression* *Statement* ⟨lookahead not else⟩
384 if *ParenExpression* *Statement* else *Statement*

SwitchStatement

385 switch *ParenExpression* { *CaseClauses*$_{\text{opt}}$ *DefaultClause*$_{\text{opt}}$ }

CaseClauses

386 case *Expression* : *Directives*$_{\text{opt}}$ *CaseClauses*$_{\text{opt}}$

DefaultClause

387 default : *Directives*$_{\text{opt}}$

WhileStatement

388 while *ParenExpression* *Statement*

DoStatement

389 do *Statement* while *ParenExpression* ;

ForStatement [20]

390 for (*ForInitializer*$_{\text{opt}}$; *Expression*$_{\text{opt}}$; *Assignments*$_{\text{opt}}$) *Statement*

[20] AS4 drops for-in and for-each-in statements. They may be reintroduced later as special cases of a general iteration construct.

ForInitializer

391 *Assignments*
392 *VariableDefinitionKind* *VariableBindings*

ContinueStatement

393 continue ;
394 continue Identifier ;

BreakStatement

395 break ;
396 break Identifier ;

ReturnStatement

397 return ;
398 return *Expression* ;

ThrowStatement

399 throw *Expression* ;

TryStatement

400 try *Statement* *CatchClauses* ⟨lookahead not finally⟩
401 try *Statement* finally *Statement*
402 try *Statement* *CatchClauses* finally *Statement*

CatchClauses

403 *CatchClause* *CatchClauses*$_{\mathtt{opt}}$

CatchClause

404 `catch` `(` *TypedBinding* `)` *Block*

SuperStatement

405 `super` *Arguments* `;`

DeferStatement [21]

406 `defer` *Statement*

[21] AS4 introduces `let`, and correspondingly stricter restrictions for enforcing its semantics, to replace AS3's `const`. To recover some of the expressiveness lost due to these restrictions as applied in constructors, AS4 also introduces defer statements.

3.3.5 Definitions

VariableDefinition

407 *VariableDefinitionKind* *VariableBindings* `;`

VariableDefinitionKind

408 `let`

409 `var`

VariableBindings

410 *VariableBinding*

411 *VariableBindings* `,` *VariableBinding*

VariableBinding

412 *TypedBinding* *VariableInitialization*$_{\mathtt{opt}}$

VariableInitialization

413 `=` *Expression*

FunctionDefinition

414 `function` *AccessorKind*$_{\mathtt{opt}}$ Identifier *OptionalFunctionBody*

Attribute

415 `native`

416 `final`

417 `override`

Static

418 `static`

AccessControl

419 `public`

420 `private`

421 `protected`

422 `internal`

AccessorKind

423 `get`
424 `set`

OptionalFunctionBody

425 *FunctionBody*
426 *FunctionSignature* `;`

ClassDefinition

427 `class` Identifier *ClassBody*

ClassInheritance

428 `extends` *NominalType*
429 `implements` *NominalTypes*
430 `extends` *NominalType* `implements` *NominalTypes*

NominalTypes

431 *NominalType*
432 *NominalTypes* `,` *NominalType*

ClassBody

433 *ClassInheritance*$_{\texttt{opt}}$ `{` *ClassDirectives* `}`

InterfaceDefinition

434 `interface` Identifier *InterfaceBody*

InterfaceInheritance

435 `extends` *NominalTypes*

InterfaceBody

436 *InterfaceInheritance*$_{\texttt{opt}}$ `{` *InterfaceDirectives* `}`

3.3.6 Directives

Configurations [22]

437 `#` *Identifier* = *Expression* `;` *Configurations*$_{\texttt{opt}}$

[22] In AS3, configuration constants were defined and used with special namespaces. With the removal of namespaces in AS4, a new syntactic mechanism is introduced for those purposes, and the configuration constants are restricted to be booleans.

ConfigurationExpression

438 `#` *Expression*

Directives

439 *ConfigurationExpression*$_{\texttt{opt}}$ *FunctionDefinition* *Directives*$_{\texttt{opt}}$
440 *ConfigurationExpression*$_{\texttt{opt}}$ *VariableDefinition* *Directives*$_{\texttt{opt}}$
441 *Statement* *Directives*$_{\texttt{opt}}$
442 *ConfigurationExpression* `{` *Directives* `}` *Directives*$_{\texttt{opt}}$

ClassDirectives

443 *ConfigurationExpression*$_{\mathtt{opt}}$ *Metadata*$_{\mathtt{opt}}$ *Modifiers*$_{\mathtt{opt}}$ *FunctionDefinition* *ClassDirectives*$_{\mathtt{opt}}$
444 *ConfigurationExpression*$_{\mathtt{opt}}$ *Metadata*$_{\mathtt{opt}}$ *Modifiers*$_{\mathtt{opt}}$ *VariableDefinition* *ClassDirectives*$_{\mathtt{opt}}$
445 *Static*$_{\mathtt{opt}}$ *Block* *ClassDirectives*$_{\mathtt{opt}}$
446 *ConfigurationExpression* `{` *ClassDirectives* `}` *ClassDirectives*$_{\mathtt{opt}}$

InterfaceDirectives

447 *ConfigurationExpression*$_{\mathtt{opt}}$ *Metadata*$_{\mathtt{opt}}$ *FunctionDefinition* *InterfaceDirectives*$_{\mathtt{opt}}$
448 *ConfigurationExpression* `{` *InterfaceDirectives* `}` *InterfaceDirectives*$_{\mathtt{opt}}$

Metadata [23]

449 `@` *ObjectInitializer* *Metadata*$_{\mathtt{opt}}$

23 AS4 introduces new syntax for metadata to replace AS3's syntax. The new syntax reuses the syntax for object initializers, and as such metadata is typechecked.

PackageName

450 Identifier
451 *PackageName* `.` Identifier

Import

452 `import` *PackageName* `. * ;`
453 `import` *PackageName* `.` Identifier `;`

Imports

454 *Import*
455 *ConfigurationExpression* `{` *Imports* `}` *Imports*$_{\mathtt{opt}}$

Modifiers

456 *Static* *Modifiers*$_{\mathtt{opt}}$
457 *Attribute* *Modifiers*$_{\mathtt{opt}}$
458 *AccessControl* *Modifiers*$_{\mathtt{opt}}$

TypeDefinitions

459 *ConfigurationExpression*$_{\mathtt{opt}}$ *Metadata*$_{\mathtt{opt}}$ *Modifiers*$_{\mathtt{opt}}$ *ClassDefinition* *TypeDefinitions*$_{\mathtt{opt}}$
460 *ConfigurationExpression*$_{\mathtt{opt}}$ *Metadata*$_{\mathtt{opt}}$ *AccessControl*$_{\mathtt{opt}}$ *InterfaceDefinition* *TypeDefinitions*$_{\mathtt{opt}}$

Unit

461 `package` *PackageName*$_{\mathtt{opt}}$ `{` *Configurations*$_{\mathtt{opt}}$ *Imports*$_{\mathtt{opt}}$ *TypeDefinitions* `}`

Units

462 *Unit* *Units*$_{\mathtt{opt}}$

Program [24]

463 *Units*

24 Unlike in AS3, where top-level definitions could include those for variables and functions, in AS4 the only top-level definitions are for classes and interfaces.

Chapter 4

Pruning

The syntax trees obtained by scanning and parsing, as described in the previous chapter, undergo pruning in order to obtain a syntactically valid program. This chapter describes pruning, which involves the processes of unit configuration and enforcement of syntactic restrictions.

4.1 Unit Configuration

4.1(1) Unit configuration proceeds by traversing a parsed unit in textual order, and as the traversal progresses, building a map from identifiers to boolean values, and simplifying syntax trees that are guarded by expressions involving such identifiers that evaluate to such boolean values.

4.1(2) *Expression*s in *Configurations* and *ConfigurationExpression*s must be composed of *Identifier*s, `true`, `false`, `!`, `&&`, and `||`. Such *Expressions* evaluate only to boolean values.

4.1(3) A syntax tree nested by some *Configurations*, of the form `#` *Identifier* = *Expression*, is processed as follows. The *Expression* is evaluated to a boolean value, possibly by looking up the map, and the *Identifier* is then mapped to that boolean value.

4.1(4) A syntax tree guarded by a *ConfigurationExpression* of the form `#` *Expression* is processed as follows. The *Expression* is evaluated to a boolean value, possibly by looking up the map. If the value is `false`, the syntax tree is eliminated. If the value is `true`, the syntax tree is retained but the guard, along with any pair of braces `{` and `}` that it introduces, is eliminated. Otherwise, an error is reported.

4.2 Enforcement of Syntactic Restrictions

Definition 4.2.1 (Global context)**.** A syntax tree is in a *global context* if it is nested by a *Program* without crossing a *ClassBody*, *InterfaceBody*, or *FunctionBody*.

Definition 4.2.2 (Class context)**.** A syntax tree is in a *class context* if it is nested by a *ClassBody* without crossing a *FunctionBody*.

Definition 4.2.3 (Interface context)**.** A syntax tree is in an *interface context* if it is nested by an *InterfaceBody*.

Definition 4.2.4 (Constructor)**.** A *constructor* is a *FunctionDefinition* that is in a class context, and whose name has an identifier that matches the identifier of that class.

Definition 4.2.5 (Getter/Setter)**.** A *getter* is a *FunctionDefinition* whose *AccessorKind* is `get`. A *setter* is a *FunctionDefinition* whose *AccessorKind* is `set`.

Definition 4.2.6 (Result type)**.** The *result type* of a *FunctionInitializer* or *FunctionDefinition* that has a *ResultType* is that *ResultType*. The *result type* of a *FunctionInitializer* or *FunctionDefinition* that does not have a *ResultType* is missing.

Definition 4.2.7 (Mark)**.** Any *AccessControl*, *Attribute*, or the keyword `static` preceding a *ClassDefinition*, *InterfaceDefinition*, *FunctionDefinition*, or *VariableDefinition* is said to *mark* it.

Definition 4.2.8 (Bodyless)**.** A *FunctionDefinition* is *bodyless* if it does not have a *FunctionBody*.

Definition 4.2.9 (Return expression)**.** A *FunctionInitializer* or *FunctionDefinition* has a *return expression* if a *ReturnStatement* is nested by it without crossing another *FunctionBody*, and the *ReturnStatement* has an *Expression*.

Definition 4.2.10 (Iterator statement)**.** An *iterator* statement is a *WhileStatement*, a *DoStatement*, a *ForStatement*, or a *LabeledStatement* whose *Statement* is an iterator statement.

The following side conditions must be satisfied to ensure that a syntax tree is in the language.

4.2.1 Class Definitions

4.2(1) The only *AccessControl*s that may mark a *ClassDefinition* are `public` and `internal`.

4.2(2) The only *Attribute* that may mark a *ClassDefinition* is `final`.

4.2(3) A *ClassDefinition* must not be marked `static`.

4.2(4) A particular *Modifier* must not mark a *ClassDefinition* more than once.

4.2.2 Interface Definitions

4.2(5) The only *AccessControl*s that may mark an *InterfaceDefinition* are `public` and `internal`.

4.2.3 Function Definitions

4.2(6) A *FunctionDefinition* that is marked `static` must not be marked by `final` or `override`.

4.2(7) A particular *Modifier* must not mark a *FunctionDefinition* more than once.

4.2(8) A *FunctionDefinition* is bodyless if and only if it is in an interface context or is marked `native`.

4.2.3.1 Getters and Setters

4.2(9) A getter or setter may appear only in a class context or interface context.

4.2(10) The *FunctionSignature* of a getter must not have *Parameters*.

4.2(11) The result type of a getter must not be `void`.

4.2(12) The *FunctionSignature* of a setter must have *Parameters*, which must be exactly one *Parameter*.

4.2(13) The result type of a setter must be `void` or missing.

4.2.3.2 Constructors

4.2(14) A constructor must not be a getter or setter.

4.2(15) No *Attribute* may mark a constructor.

4.2(16) A constructor must not be marked `static`.

4.2(17) The result type of a constructor must be `void` or missing.

4.2.4 Variable Definitions

4.2(18) A *VariableDefinition* must not be marked by an *Attribute*.

4.2(19) No *Modifier* may mark a *VariableDefinition* more than once.

4.2.5 Super Statements

4.2(20) A *SuperStatement* must be the first *Directive* in the *FunctionBody* of a constructor.

4.2.6 Defer Statements

4.2(21) A *DeferStatement* must be the last *Directive* in the *FunctionBody* of a constructor.

4.2.7 Labeled Statements

4.2(22) The label of a *LabeledStatement* must not be the label of another *LabeledStatement* that nests it without crossing a *FunctionBody*.

4.2.8 Break Statements

4.2(23) A *BreakStatement* must be nested by a *WhileStatement*, a *DoStatement*, *ForStatement*, a *SwitchStatement*, or a *LabeledStatement* without crossing a *FunctionBody*.

4.2(24) A *BreakStatement* must carry a label if it is not nested by a *WhileStatement*, a *DoStatement*, a *ForStatement*, or a *SwitchStatement*.

4.2(25) The label of a *BreakStatement* must be the label of a *LabeledStatement* that nests it without crossing a *FunctionBody*.

4.2.9 Continue Statements

4.2(26) A *ContinueStatement* must be nested by an iterator statement without crossing a *FunctionBody*.

4.2(27) The label of a *ContinueStatement* must be the label of a *LabeledStatement* that nests it without crossing a *FunctionBody*, and is an iterator statement.

4.2.10 Return Statements

4.2(28) A *ReturnStatement* must be nested by a *FunctionInitializer* or *FunctionDefinition*.

4.2(29) A *FunctionInitializer* or *FunctionDefinition* must have a return expression if and only if its result type is not `void` or missing.

4.2(30) Any return statement in any constructor or setter must not have a return expression.

4.2(31) A getter must have *ReturnStatement*s that have return expressions.

4.2.11 Expressions

4.2(32) The keyword `this` must be nested by a *VariableDefinition* or *FunctionDefinition* that is in a class context and is not marked `static`.

4.2(33) A *SuperExpression* must be nested by a *VariableDefinition* or *FunctionDefinition* that is in a class context and is not marked `static`, without crossing a *FunctionBody*.

4.2(34) Any *Expression*s in the *ObjectInitializer*s of *Metadata* must be `null`, or `true`, or `false`, or a *NumericLiteral*, or a *StringLiteral*, or a *TypeExpression*.

Chapter 5

Canonicalization

This chapter specifies the equivalence of certain forms of syntax trees with other, canonical forms. Considering only canonical forms of syntax trees helps simplify the description of compilation and execution algorithms in the sequel: by narrowing the set of syntactic forms under consideration, the semantic rules become more concise. Importantly, an implementation does not need to canonicalize syntax trees: the semantic rules for non-canonical forms of syntax trees can be readily and unambiguously derived from those for their equivalent canonical forms.

5.1 Canonical Forms

5.1(1) A *VariableDefinition* with multiple *VariableBinding*s is treated as if it were a sequence of *VariableDefinition*s, each with the same *VariableDefinitionKind* and a single *VariableBinding*.

5.1(2) An *Assignments* with multiple *Assignment*s is treated as if it were a sequence of *Statement*s, each with a single *Assignment*.

5.1(3) A missing *AccessControl* is treated as if it were `internal`.

5.1(4) If a *ClassDefinition* whose *Identifier* is C does not nest a constructor, a *FunctionDefinition* is added to the the *ClassDefinition*, which is not marked `static`, has *AccessAttribute* `public`, whose *Identifier* is C, whose *FunctionBody* has an empty *Directives*, and whose *FunctionSignature* has empty *Parameters* and whose *ResultType* is `void`.

5.1(5) If a constructor does not nest a *SuperStatement*, a *SuperStatement* is added to the top of the *Directives* of its *FunctionBody*, whose *Arguments* is empty.

5.1(6) Any *VariableInitializer*s corresponding to *VariableDefinition*s not marked `static` in a *ClassDefinition* are treated as if they were *AssignmentStatement*s, and alongwith any *Block*s in the *ClassDefinition* that are not marked `static`, are moved in order to the top of the constructor just below the *SuperStatement*.

5.1(7) A *FunctionDefinition* called the *static constructor* is added to a *ClassDefinition* named C whose *Identifier* is **static** :: C, which is marked `static`, has *AccessAttribute* `public`, has no *Parameter*s, whose return type is `void`, and whose *Identifier* is C. Any *VariableInitializer*s corresponding to *VariableDefinition*s marked `static` in the *ClassDefinition* are treated as if they were *AssignmentStatement*s, and alongwith any *Block*s in the *ClassDefinition* that are marked `static`, are moved in order to the *FunctionBody* of the static constructor.

5.1(8) Constructors are moved to the top of the instance scopes of their class bodies, and static constructors are moved to the top of the static scopes of their class bodies.

5.1(9) A *ForStatement* of the form `for ( ` *ForInitializer*$_{\texttt{opt}}$ `;` *Expression*$_{\texttt{opt}}$ `;` *Assignments*$_{\texttt{opt}}$ `)` *Statement* is treated as if it were `{` *ForInitializer*$_{\texttt{opt}}$ `; while (` *Expression* `) {` *Statement Assignments*$_{\texttt{opt}}$ `} }`.

5.1(10) A *DoStatement* of the form `do` *Statement* `while` *ParenExpression* is treated as if it were *Statement* `while (` *Expression* `)` *Statement*.

5.1(11) An *IfStatement* of the form `if` *ParenExpression Statement* is treated as if it were `if` *ParenExpression Statement* `else ;`.

5.1(12) *Metadata* is treated as if it were an *ArrayInitializer* of the form `new []Object{`$e_1, \ldots, e_n$`}`, where each e_i ($i \in 1..n$) is of the form `new` *ObjectInitializer* derived from a corresponding syntax tree of the form `@`*ObjectInitializer* in the *Metadata*.

5.1(13) A *NewExpression* of the form `new` *NominalType* is treated as if it were `new` *NominalType* `()`. A *NewExpression* of the form `new` *NominalType Arguments* `{x1 = e1,...,xn = en}` is treated as if it were the syntax tree `function (o :`*NominalType*`) {o.x1 = e1,...,o.xn = en} (new` *NominalType Arguments*`)`.

5.1(14) A *FunctionInitializer* of the form `function` Identifier *FunctionBody* is treated as if it were `function () { function` Identifier *FunctionBody*`; return` Identifier`; } ()`.

5.1(15) An assignment of the form *ReferenceExpression* *binop*= *Expression*, where *ReferenceExpression* is an *Identifier* or of the form *NominalType MemberOperator*, is treated as if it were *ReferenceExpression* = *ReferenceExpression* *binop* *Expression*. An assignment of the form *ReferenceExpression* *binop*= *Expression*, where *ReferenceExpression* is of the form *BaseExpression MemberOperator*, is treated as if it were `let x =` *BaseExpression*`; x` *MemberOperator* = `x` *MemberOperator* *binop* *Expression*. An assignment of the form *ReferenceExpression* *binop*= *Expression*, where *ReferenceExpression* is of the form *BaseExpression* `[` *index* `]`, is treated as if it were `let x =` *BaseExpression*`; let y =` *index*`; x[y] = x[y]` *binop* *Expression*.

5.1(16) A prefix operation of the form *prefixop* *ReferenceExpression*, where *ReferenceExpression* is an *Identifier* or of the form *NominalType MemberOperator*, is treated as if it were `function() {` *ReferenceExpression* = *ReferenceExpression* *binop* `1; return` *ReferenceExpression*`; } ()`, where *binop* is the binary operation corresponding to *prefixop*. A prefix operation of the form *prefixop* *ReferenceExpression*, where *ReferenceExpression* is of the form *BaseExpression MemberOperator*, is treated as if it were the syntax tree `function() { let x =` *BaseExpression*`; x` *MemberOperator* = `x` *MemberOperator* *binop* `1; return x` *MemberOperator*`; } ()`, where *binop* is the binary operation corresponding to *prefixop*. A prefix operation of the form *prefixop* *ReferenceExpression*, where *ReferenceExpression* is of the form *BaseExpression* `[`*index*`]`, is treated as if it were the syntax tree `function() { let x =` *BaseExpression*`; let y =` *index*`; x[y] = x[y]` *binop* `1; return x[y]; } ()`, where *binop* is the binary operation corresponding to *prefixop*.

5.1(17) A postfix operation of the form *ReferenceExpression* *postfixop*, where *ReferenceExpression* is an *Identifier* or of the form *NominalType MemberOperator*, is treated as if it were `function() { let y =` *ReferenceExpression*`;` *ReferenceExpression* `= y` *binop* `1; return y; } ()`, where *binop* is the binary operation corresponding to *postfixop*. A postfix operation of the form *ReferenceExpression* *postfixop*, where *ReferenceExpression* is of the form *BaseExpression MemberOperator*, is treated as if it were the syntax tree `function() { let x =` *BaseExpression* `; let y = x`*MemberOperator*`; x` *MemberOperator* = `y` *binop* `1; return y; } ()`, where *binop* is the binary operation corresponding to *postfixop*. A postfix operation of the form *ReferenceExpression* *postfixop*, where *ReferenceExpression* is of the form *BaseExpression* `[`*index*`]`, is treated as if it were the syntax tree `function() { let x =` *BaseExpression*`; let z =` *index*`; let y = x[z]; x[z] = y` *binop* `1; return y; } ()`, where *binop* is the binary operation corresponding to *postfixop*.

5.1(18) A unary plus operation of the form `+`*Expression* is treated as if it were *Expression*. A unary minus operation of the form `-`*Expression* is treated as if it were `0 -` *Expression*.

5.2 Non-Canonical Forms

5.2(1) As a consequence of canonicalization, the following syntactic forms are assumed to be encoded away, and are not considered when specifying semantic rules in the sequel:

1. *VariableDefinition*s with multiple *VariableBinding*s
2. *Assignments* with multiple *Assignment*s
3. missing *AccessControl*s
4. missing constructors in *ClassDefinition*s
5. missing *SuperStatement*s
6. *VariableInitializer*s of *VariableDefinition*s, marked `static` or not, in *ClassDefinitions*
7. *ForStatement*s
8. *DoStatement*s
9. *IfStatement*s without `else` *Statement*s
10. *Metadata*
11. *NewExpression*s without *Arguments*, or with *FieldValuePairs*
12. *FunctionInitializer*s with *Identifier*s
13. compound *Assignment*s of the form *op*`=`
14. prefix and postfix operations `--` and `++`
15. unary `-` and `+` operations

Part III

Compilation

Chapter 6

Resolution and Lexical Environments

This chapter describes the process of deriving compile-time information for types defined by a program and the set of libraries it is compiled against. The compile-time information is recorded as *lexical environments*. Along the way, identifiers are lexically resolved to various kinds of references: to local variables and functions, to instance members, to static members, and to package-qualified types.

6.1 References

6.1(1) References to definitions are looked up for various purposes at compile time and run time. Some references correspond to syntactic forms, whereas others are generated during *resolution*, as described below.

Definition 6.1.1 (Reference)**.** A *reference* is either an Identifier, or of the form *NominalType*, or of the form `super.`Identifier, or of the form *NominalType*`.`Identifier, or of the form **static**`.`Identifier, or of the form *Expression*`.`Identifier (member reference), or of the form *Expression*$_1$`[`*Expression*$_2$`]` (index reference).

6.1.1 Scopes

6.1(2) Scopes are identified with specific forms of syntax trees, and can be nested (like syntax trees).

6.1(3) A *ClassBody* is conceptually partitioned into a pair of scopes: a *static scope* and an *instance scope*. The static scope contains variables and functions marked `static`, including the static constructor. The instance scope contains variables and functions not marked `static`, including the (instance) constructor.

Definition 6.1.2 (Scope)**.** A *scope* is either a global scope or a local scope. A global scope is the instance scope or the static scope of a *ClassBody*, or an *InterfaceBody*. A local scope is a *FunctionBody*, a *CatchClause*, or a *Block*.

6.1.2 Lexical Environments

6.1(4) Every scope is associated with a lexical environment, which is used to store lexical bindings for definitions in that scope and to look up names in that scope. Lexical environments provide information at compile time as well as run time (upon regeneration).

Definition 6.1.3 (Lexical environment and lexical bindings)**.** A *lexical environment* is a container for lexical bindings. A *lexical binding* a (*name*, *Definition*) pair.

6.1(5) In addition, there is a *global lexical environment* that maps the fully qualified names for types (classes and interfaces) that are defined in the set of libraries and the *Program* to their definitions, with no name conflicts. The name of such a defined type is fully qualified as P.id or **internal** ::P.id according as whether the definition is marked `public` or not, where P is the name of the package in which it is defined (which is empty if no such package is specified, meaning the default package) and id is the identifier in that definition.

6.2 Resolution

6.2(1) At compile time, *ReferenceExpression*s that are Identifiers are transformed to references, as follows. Let id be an identifier that is nested by a scope *Scope* without crossing another scope. Then id is transformed to the reference *ref*, returned by the following computation.

1. If there is a lexical binding of the form (id, *def*) in the lexical environment associated with *Scope*, return id. Furthermore, if the enclosing scope *Scope'* of *def* nests *Scope*, then record the fact that id is resolved in all scopes that nest *Scope* without crossing *Scope'*.
2. If *Scope* is enclosed by the instance scope of a *ClassDefinition* that has a lexical binding of the form (id, *def*), then return the member reference `this`.id.
3. If *Scope* is enclosed by the static scope of a *ClassDefinition* that has a lexical binding of the form (id, *def*), then return the lexical reference **static**.id.
4. If id appears where a *NominalType* is expected, it must resolve to a type (class or interface). If there is a unique P such that either a type of the form **internal** ::P.id in the global lexical environment and P is the package name of the enclosing unit, or a type of the form P.id is in the global lexical environment and either P is empty, or P is the package name of the enclosing unit, or there is an import of the form P.id or P.`*` in the enclosing unit, then return the fully qualified name of the type.
5. Otherwise, report an error.

6.3 Building Lexical Environments

6.3.1 Global Scopes

6.3(1) The definition corresponding to each fully qualified name in a *ClassInheritance* must be a *ClassDefinition*, and the definition corresponding to each fully qualified name in a *InterfaceInheritance* must be a *InterfaceDefinition*, otherwise an error is reported.

6.3(2) A class must not recursively extend itself, and an interface must not recursively extend itself, otherwise an error is reported.

6.3(3) At compile time, definitions of classes and interfaces are visited in topologically-sorted order following the `extends` relation.

6.3(4) Upon visiting an *InterfaceDefinition*, any types appearing in its metadata must be resolved. The lexical environment of an interface body contains lexical bindings for the names and definitions of functions in it, as well as the lexical bindings of any interface it extends. These involve the resolution of types in their signatures and metadata.

1. A name must correspond to either a function without an accessor kind, or a getter or setter.
2. If there are multiple definitions with the same name in this scope, an error is reported unless they form a getter/setter pair.

3. If there are multiple lexical bindings with the same name for a function, and the function does not have an accessor kind or is a getter or setter with the same accessor kind, then their signatures must match, and only one is retained.

4. If there is a getter/setter pair, then their signatures must be complementary.

6.3(5) Upon visiting a *ClassDefinition*, any types appearing in its metadata must be resolved. The class extended by the class must not be marked `final`.

1. The lexical environment of the static scope of a class body contains lexical bindings for the names and definitions of variables and functions in it, including the static constructor, as well as any non-conflicting lexical bindings (i.e., with distinct identifiers) of the static scope for any class it extends. These involve the resolution of the types and metadata of the variables and the types in the signatures and the metadata of the functions that appear in it.

    (a) If there are multiple variables or functions with the same name in this scope, an error is reported unless they form a getter/setter pair.

    (b) If there is a getter/setter pair, then their signatures must be complementary.

2. The lexical environment of the instance scope of a class body contains lexical bindings for the names and definitions of variables and functions in it, including the (instance) constructor, as well as the lexical bindings of the instance scope for any class it extends. These involve the resolution of the types and metadata of the variables and the types in the signatures and the metadata of the functions that appear in it.

    (a) There must not be any conflicts between the lexical bindings of the instance scope and the static scope of the class body.

    (b) A name must correspond to either a variable, or a function without an accessor kind, or a getter or setter.

    (c) If there are multiple definitions with the same name in this scope, an error is reported unless they form a getter/setter pair.

    (d) For any variable, if there are two lexical bindings with the same name, an error is reported.

    (e) If there are two lexical bindings with the same name for a function:

        i. If there is a getter/setter pair, then their signatures must be complementary.

        ii. For a function that does not have an accessor kind, or for a getter or setter with the same accessor kind, their signatures and access controls must match, and only the one defined in this scope is retained. Furthermore, only in such a case may the function defined in this scope be marked `override`. Finally, the function not defined in this scope must not be marked `final`.

    (f) For every function in the lexical binding of every interface implemented by the class, there must be a lexical binding for a function in the instance scope of the class, that has a matching signature and is marked `public`.

6.3(6) For any other scope *Scope*, we build the lexical environment of that *Scope* as described in the next section.

6.3.2 Local Scopes

6.3(7) Building lexical environments for local scopes involves synthesizing lexical bindings for local variables and local functions in the *Program*. (Lexical bindings for classes and interfaces, as well as their member variables and member functions, are synthesized as described in the previous section.)

6.3(8) Lexical environments for local scopes are built by processing them in depth-first (textual) order.

6.3(9) Upon visiting a *FunctionBody*, a lexical environment is associated with the scope that initially contains lexical bindings for the parameters (see below), and any non-conflicting lexical bindings (i.e., with distinct identifiers) of the enclosing scope. Each parameter introduces a typed binding, and thereby corresponds to a *VariableDefinition* (treated as if it had *VariableDefinitionKind* `let`). Thus, the lexical binding for it is of the form (id, *def*), where id is the identifier of the typed binding, and *def* is the *VariableDefinition* itself. The identifiers added must be distinct; otherwise an error is reported.

6.3(10) Upon visiting a *CatchClause*, a lexical environment is associated with the scope that initially contains a lexical binding for the *TypedBinding* of the *CatchClause* (see below), and any non-conflicting lexical bindings (i.e., with distinct identifiers) of the enclosing scope. The *TypedBinding* corresponds to a *VariableDefinition* (treated as if it had *VariableDefinitionKind* `let`). Thus, the lexical binding for it is of the form (id, *def*), where id is the identifier of the *TypedBinding*, and *def* is the *VariableDefinition* itself.

6.3(11) Upon visiting a *Block*, a lexical environment is associated with the scope that initially contains the lexical bindings of the enclosing scope.

6.3(12) Upon visiting a *VariableDefinition* *def*, a lexical binding of the form (id, *def*) is added to the lexical environment associated with the enclosing scope, where id is the identifier of *def*, unless id has already been resolved in the enclosing scope, whereupon an error is reported. The fact that id has been resolved in the enclosing scope is now recorded, and if there is an existing lexical binding with the same identifier in the lexical environment, then it is removed.

6.3(13) Upon visiting a *FunctionDefinition* *def*, a lexical binding of the form (id, *def*) is added to the lexical environment associated with the enclosing scope, where id is the identifier of *def*, unless id has already been resolved in the enclosing scope, whereupon an error is reported. The fact that id has been resolved in the enclosing scope is now recorded, and if there is an existing lexical binding with the same identifier in the lexical environment, then it is removed.

Chapter 7

Inference of Types and Constants

This chapter describes static verification of a program given compile-time information encoded by lexical environments. Along the way, missing types are inferred, compile-time constants are propagated, and implicit coercions are made explicit. The resulting program can be viewed as an executable that is ready to be executed.

7.1 Interpretation of Missing Types

7.1(1) Missing *Type*s for *VariableDefinition*s that appear in local contexts or are marked `let`, and missing *ResultType*s, are replaced by fresh type variables. These type variables are eventually replaced by *Type*s computed by type inference. Any other missing *Type*s are considered to be `*`, and missing *ResultType*s of functions that do not have *ReturnStatement* with *Expression*s are assumed to be `void`.

7.2 Types

7.2(1) The various kinds of types T are as follows:

1. `*` (for dynamic values)
2. value types `int`, `uint`, `long`, `ulong`, `double`, `float`, `byte`, `bool`
3. C (for instances of the class C)
4. I (for instances of classes that implement the interface I)
5. $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow T$ (for functions with the sequence of non-optional parameter types $T_1, \ldots, T_i$, the sequence of optional parameter types $T_{i+1}, \ldots, T_{i+j}$, and the result type T)
6. $\{$`null`$\}$ (for `null`)
7. $\{N\}$ (for numeric literals N)
8. `void` (for *no value*)

7.2(2) In addition, at compile-time types include:

1. *type variables* X
2. *type operations* that involve type variables:

(a) $T.x$ denoting the type of member x for types T of objects with member x

(b) $\mathsf{elem}(T)$ denoting the element type for types T of arrays and array lists

(c) $\mathsf{param}_k(T)$ denoting the type of the k^{th} parameter for types T of functions with the k^{th} parameter

(d) $\mathsf{return}(T)$ denoting the return type for types T of functions

(e) $\mathsf{LUB}(T_1, T_2)$ denoting the least upper bound for types T_1 and T_2

(f) $\mathsf{add}(T_1, T_2)$ denoting the type of addition of values of types T_1 and T_2.

Definition 7.2.1 (Type of definition)**.** The *type* of a definition def is computed as follows:

1. If def is unknown to the compiler, then return `*`.
2. If def is a *VariableDefinition*, then return its *Type*.
3. If def is a *FunctionDefinition*:
   (a) If it is a getter/setter, return its *Type* (which is the *ResultType* of a getter and the *Type* of the *Parameter* of a setter).
   (b) Otherwise, return $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow T$, where $T_{i+1}, \ldots, T_{i+j}$ are the types of the *OptionalParameter*s, and $T_1, \ldots, T_i$ are the types of the other *Parameter*s,
4. If def is a *ClassDefinition*, return `Class`.

7.3 Typing Relations

7.3(1) The following notions of subtyping, implicit coercibility, and type compatibility control, at compile time, which types of values are considered safe to store in which types of locations at run time.

Definition 7.3.1 (Subtyping)**.** Subtyping is a binary relation on types, defined by the transitive closure of the following rules:

1. Any type is a subtype of itself.
2. The type {`null`} is a subtype of any non-value (reference) type.
3. If a class C extends another class C', then C is a subtype of C'.
4. If an interface I extends another interface I', then I is a subtype of I'.
5. If a class C implements an interface I, then C is a subtype of I.
6. Any type of the form $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow T$ is a subtype of $(T'_1, \ldots, T'_{i+j}) \Rightarrow T'$ where T'_m is a subtype of T_m for all $m \in 1..i+j$, and T is a subtype of T'.

Definition 7.3.2 (Promotion, implicit coercibility, and type compatibility)**.** A numeric literal fits the first of `int`, `uint`, `long`, `ulong`, and `double` that it is in the range of.

Promotion is a binary relation on numeric types, specified by the following table (where a type heading a row is promotable to a type heading a column if the entry common to the row and column is marked $\surd$):

	byte	int	uint	long	ulong	float	double
byte	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
int		$\surd$		$\surd$			$\surd$
uint			$\surd$	$\surd$	$\surd$		$\surd$
long				$\surd$			
ulong					$\surd$		
float						$\surd$	$\surd$
double							$\surd$

In addition, the type $\{N\}$, where N is a numeric literal, is promotable to the numeric type T if the numeric literal N fits the numeric type T', and $N : T'$ is implicitly coercible to T (see below).

Type compatibility is subtyping, promotion, or implicit coercibility, where implicit coercibility is a binary relations on types, defined by the following rules:

1. Any type (other than `void`) is implicitly coercible to the type `*`.
2. The type `*` is implicitly coercible to any type (other than `void`).
3. Implicit coercibility between numeric types is specified by the following table (where a type heading a row is implicitly coercible to a type heading a column if the entry common to the row and column is marked $\surd$ or specifies a constraint to be satisfied by the value being coerced):

	byte	int	uint	long	ulong	float	double
byte	$\surd$						
int	$\geq 0, < 2^8$	$\surd$	≥ 0		≥ 0	$\geq -2^{24}, < 2^{24}$	
uint	$< 2^8$	$< 2^{31}$	$\surd$			$< 2^{24}$	
long	$\geq 0, < 2^8$	$\geq -2^{31}, < 2^{31}$	$\geq 0, < 2^{32}$	$\surd$	≥ 0	$\geq -2^{24}, < 2^{24}$	$\geq -2^{53}, < 2^{53}$
ulong	$< 2^8$	$< 2^{31}$	$< 2^{32}$	$< 2^{63}$	$\surd$	$< 2^{24}$	$< 2^{53}$
float						$\surd$	
double						$\surd$	$\surd$

Definition 7.3.3 (Member type)**.** The type operation $T.m$ is defined as follows:

1. If T is an interface whose member m is of type T', return T'.
2. If T is a class whose instance member m is of type T', return T'.
3. If T is `*`, return `*`.

Definition 7.3.4 (Element type)**.** The type operation $\mathsf{elem}(T)$ is defined as follows:

1. If T is $[]T'$ or `ArrayList` $<T>$, return T'.
2. If T is `*`, return `*`.

Definition 7.3.5 (Parameter type)**.** The type operation $\mathsf{param}_k(T)$ is defined as follows:

1. If T is a function type whose k^{th} parameter type is T', return T'.
2. If T is `*`, return `*`.

Definition 7.3.6 (Return type)**.** The type operation $\mathsf{return}(T)$ is defined as follows:

1. If T is a function type whose return type is T', return T'.
2. If T is `*`, return `*`.

Definition 7.3.7 (LUB of types)**.** The (symmetric) type operation $\mathsf{LUB}(T_1, T_2)$ is defined as follows:

1. The union of T_1 and T_2, if defined, is returned.

2. Otherwise, the LUB of two numeric types T_1 and T_2 is T_3 such that, we have T_1 and T_2 are promotable to T_3, and for any T_3' such that T_1 and T_2 are promotable to T_3', we have that T_3 is promotable to T_3'.
3. The LUB of any other pair of types is `*`.

Definition 7.3.8 (Add of types)**.** The (symmetric) type operation $\mathsf{add}(T_1, T_2)$ is defined as follows:

1. Return `String` if any of T_1 and T_2 is `String`.
2. Otherwise return $\mathsf{LUB}(T_1, T_2)$.

Definition 7.3.9 (Union of types)**.** The union of a pair of types (symmetric) is defined as follows:

1. The union of two singleton numeric types $\{N_1\}$ and $\{N_2\}$ is the first among `int`, `uint`, `long`, `ulong`, and `double` that both N_1 and N_2 can be promoted to.
2. The union of T and T is T for all types T.
3. The union of two reference types is their least common ancestor in the inheritance hierarchy.
4. The union of two function types $(S_1, \ldots, S_i, S_{i+1}?, \ldots, S_{i+j}?) \Rightarrow T$ and $(S'_1, \ldots, S'_{i'}, S'_{i'+1}?, \ldots, S'_{i+j}?) \Rightarrow T'$ is the function type $(S''_1, \ldots, S''_{i+j}) \Rightarrow T''$, where each S''_m is the intersection of S_m and S'_m for $m \in \{1, \ldots, i+j\}$, and T'' is the union of T and T'.
5. The union of any other pair of types is undefined.

Definition 7.3.10 (Intersection of types)**.** The intersection of a pair of types (symmetric) is defined as follows:

1. The intersection of T and T is T for all types T.
2. The intersection of two reference types is one of the types, so that the other is an ancestor in the inheritance hierarchy.
3. The intersection of two function types $(S_1, \ldots, S_i, S_{i+1}?, \ldots, S_{i+j}?) \Rightarrow T$ and $(S'_1, \ldots, S'_{i'}, S'_{i'+1}?, \ldots, S'_{i+j}?) \Rightarrow T'$ is the function type $(S''_1, \ldots, S''_{i+j}) \Rightarrow T''$, where each S''_m is the union of S_m and S'_m for $m \in \{1, \ldots, i+j\}$, and T'' is the intersection of T and T'.
4. The intersection of any other pair of types is undefined.

7.4 Coercions and Constraints

7.4(1) A coercion from type T_1 to type T_2 is generated by the compiler when an expression whose type is computed to be T_1 flows to a context that expects type T_2. Such coercions may involve type variables, but by type inference eventually involve only *Type*s.

7.4(2) A coercion from *Type* T_1 to *Type* T_2 is valid if T_1 is compatible with T_2.

7.4(3) A coercion from *Type* T_1 to *Type* T_2 is redundant if T_1 is a subtype of T_2; a redundant coercion can be erased.

7.4(4) A constraint on type T, specified as a set of types, is generated by the compiler when an expression whose type is computed to be T flows to a context that expects T to be in that set of types. Such constraints may involve type variables, but by type inference eventually involve only *Type*s.

7.4(5) A constraint on *Type* T is satisfied if it is in the specified set of types, or is `*`.

7.5 Constant Evaluation of Expressions

7.5(1) Constant evaluation is the process of evaluating expressions at compile time. The effect of constant evaluation is that some expressions are replaced by their constant values, and therefore have those values at run time.

7.5(2) Constant expressions are required in some contexts. In particular:

1. An expression that occurs in *Dimensions* to denote the size of an array must be a constant expression.
2. An expression that occurs in an *OptionalParameter* to denote the default value of that parameter must be a constant expression.
3. A constant expression that occurs in the *VariableInitialization* of a `let` *VariableDefinition* causes the *VariableInitialization* to have the value of that constant expression, thus making it a *constant binding*.

7.5(3) The observables of the language are primitive values, as defined below.

Definition 7.5.1 (Primitive type)**.** A primitive type is `int`, `uint`, `long`, `ulong`, `double`, `float`, `byte`, `bool`, and `String`.

Definition 7.5.2 (Primitive value)**.** A primitive value is a value of primitive type.

Definition 7.5.3 (Constant *VariableDefinition*, constant value)**.** A *VariableDefinition* is constant if it is marked `let` and its *VariableInitializer* has a primitive value. The constant value of a lexical binding for this *VariableDefinition* is that primitive value coerced to the type of the *VariableDefinition*.

7.6 Computing Types and Constant Values

7.6(1) A class *depends* on another class if it is a subclass of that class, or if its static constructor refers to that class.

7.6(2) No class should recursively depend on itself, otherwise an error is reported.

7.6(3) At compile time, types and constant values are computed by visiting classes in topologically-sorted order following the *depends* relation, and in those classes, visiting local scopes in the same order as prescribed for building lexical environments.

7.6.1 References

7.6(4) The type, and the optional constant value, of a reference is computed as follows:

1. If the reference is an identifier, look up the identifier in the lexical environment, yielding a definition; return the type, and the constant value if it exists, of the definition.
2. If the reference is of the form **static**`.m`, look up `m` in the lexical environment of the static scope of the enclosing class definition, yielding a definition; return the type, and the constant value if it exists, of the definition.
3. If the reference is of the form `(P.C).x`, look up `x` in the lexical environment of the static scope of the class definition mapped to `P.C` in the global lexical environment, yielding a definition; assert that the definition appears in the class body of `P.C`, return the type, and the constant value if it exists, of the definition.

4. If the reference is of the form `super.m`, look up `m` in the lexical environment of the instance scope of the base class of the enclosing class definition, yielding a function definition; return the type of the definition.
5. If the reference is of the form `this.m`, look up `m` in the lexical environment of the instance scope of the enclosing class definition, yielding a definition; return the type, and the constant value if it exists, of the definition.
6. If the reference is of the form `o.x`, compute the type T of o, and return T.`x`.
7. If the reference is of the form `c[e]`, compute the type I of e and coerce e to `int`. Also compute the type T of c. Return $\mathsf{elem}(T)$.

7.6.2 Primary Expressions

7.6.2.1 Null Literal

7.6(5) The type and constant value of `null` is computed as follows:

1. Return {`null`} and the `null` literal.

7.6.2.2 Boolean Literal

7.6(6) The type and constant value of `true` or `false`, is computed as follows:

1. Return `bool` and the boolean literal.

7.6.2.3 Numeric Literal

The type and constant value of a numeric literal N is computed as follows:

1. Return $\{N\}$ and N.

7.6.2.4 String Literal

The type and constant value of a string literal is computed as follows:

1. Return `String` and the string literal.

7.6.2.5 Regular Expression Literal

The type of a regular expression literal is computed as follows:

1. Return `RegExp`.

7.6.2.6 Array Initializer

7.6(7) The type of an *ArrayInitializer* of the form `new [e]T` is computed as follows:

1. Let the type and constant value of e be N and n.
2. Coerce $n : N$ to `uint` and then to `int`.

3. Return `[]T`.

7.6(8) The type of an *ArrayInitializer* of the form `new []T{...}` is computed as follows:

1. Let the *Expression*s of its *ArrayElement*s be $\mathsf{exp}_1, \ldots, \mathsf{exp}_k$.
2. For each $\ell \in \{1, \ldots, k\}$, compute the type and optional constant value T_ℓ and v_ℓ of exp_ℓ, and coerce them to T.
3. Return `[]T`.

7.6.2.7 Function Initializer

7.6(9) The type of a *FunctionInitializer* is computed as follows:

1. Let the types of the non-optional *Parameter*s in its *FunctionSignature* be $T_1, \ldots, T_i$, the types and *Expression*s of the *OptionalParameter*s in its *FunctionSignature* be $T_{i+1}, \ldots, T_{i+j}$ and $e_{i+1}, \ldots, e_{i+j}$, and its *ResultType* be T.
2. The expressions $e_{i+1}, \ldots, e_{i+j}$ must have constant values $v_{i+1}, \ldots, v_{i+j}$, and let their types be $T'_{i+1}, \ldots, T'_{i+j}$.
3. Coerce $v_{i+1} : T'_{i+1}, \ldots, v_{i+j} : T'_{i+j}$ to $T_{i+1}, \ldots, T_{i+j}$ and return $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow T$.

7.6.2.8 This Expression

7.6(10) The type of `this` is computed as follows:

1. Let the enclosing *ClassDefinition* be C.
2. Return C.

7.6.2.9 Type Expression

7.6(11) The type of `:`T is computed as follows:

1. If T is a class, return `Class`.
2. Otherwise T must be an interface, return `Interface`.

7.6.3 Call Expressions

7.6(12) The type of a *BaseExpression* with a trailing *Arguments* is computed as follows:

1. Let the *BaseExpression* preceding the *Arguments* be funexp, and the *Expression*s in the *Arguments* be $\mathsf{argexp}_1, \ldots, \mathsf{argexp}_k$.
2. Compute the type of funexp, yielding T_{fun}.
3. For each $\ell \in \{1, \ldots, k\}$, compute the type and optional constant value of argexp_ℓ, yielding T_ℓ and argexp_ℓ. Coerce $\mathsf{argexp}_\ell : T_\ell$ to $\mathsf{param}_\ell(T_{\mathit{fun}})$ for each $\ell \in \{1, \ldots, k\}$, and return $\mathsf{return}(T_{\mathit{fun}})$.

7.6.4 New Expressions

7.6(13) The type of a *NewExpression* is computed as follows:

1. Let the *NominalType* be C, and the *Expression*s in the *Arguments* be $\mathsf{argexp}_1, \ldots, \mathsf{argexp}_k$.
2. For each $\ell \in \{1, \ldots, k\}$, compute the type and optional constant value of argexp_ℓ, yielding T_ℓ and argexp_ℓ.
3. In the lexical environment of the instance scope of the class definition associated with C in the global lexical environment, let $(T'_1, \ldots, T'_i, T'_{i+1}?, \ldots, T'_{i+j}?) \Rightarrow$ `void` be the type of the constructor.
4. If $i \le k \le i + j$, then coerce $\mathsf{argexp}_1 : T_1, \ldots, \mathsf{argexp}_k : T_k$ to $T'_1, \ldots, T'_k$.
5. Return C.

7.6.5 Unary Expressions

7.6.5.1 Bitwise Not Expression

7.6(14) The type of a bitwise not expression of the form `~argexp` is computed as follows:

1. Compute the type of argexp under LexEnv, yielding T.
2. Constrain T to be integral but not `byte`.
3. If argexp at type T_1 has a constant value then let N be the result of applying `~` on it.
   (a) If T_1 is a singleton numeric literal, return N and $\{N\}$.
   (b) Otherwise, return N and T_1.
4. Otherwise, return T_1.

7.6.5.2 Logical Not Expression

7.6(15) The type of a logical not expression of the form `!argexp` is computed as follows:

1. Compute the type T of argexp under LexEnv.
2. Coerce argexp : T to `bool`.
3. Return `bool`, and if the expression at type `bool` has a constant value then also return the result of applying `!` on it.

7.6.6 Binary expressions

7.6.6.1 Multiplicative Expression, Subtract Expression, Relational Expression

7.6(16) The type of a multiplicative, subtract, or relational expression of the form exp_1 `*` exp_2, exp_1 `/` exp_2, exp_1 `%` exp_2, exp_1 `-` exp_2, exp_1 `<` exp_2, exp_1 `<=` exp_2, exp_1 `>` exp_2, or exp_1 `>=` exp_2 is computed as follows:

1. Compute the type of exp_1 under LexEnv, yielding T_1.
2. Compute the type of exp_2 under LexEnv, yielding T_2.
3. Constrain T_1 and T_2 to be numeric but not both `byte`.

4. Coerce $\texttt{exp}_1 : T_1$ and $\texttt{exp}_2 : T_2$ to $\mathrm{LUB}(T_1, T_2)$.
5. If the expressions at type $\mathrm{LUB}(T_1, T_2)$ have constant values then let N be the result of applying `*`, `/`, `%`, `-`, `<`, `<=`, `>`, or `>=` on them.
   (a) If T_1 and T_2 are singleton numeric literals, return N and $\{N\}$.
   (b) Otherwise, return N and $\mathrm{LUB}(T_1, T_2)$.
6. Otherwise, return $\mathrm{LUB}(T_1, T_2)$.

7.6.6.2 Add Expression

7.6(17) The type of an add expression of the form $\texttt{exp}_1$ `+` $\texttt{exp}_2$ is computed as follows:

1. Compute the type of $\texttt{exp}_1$ under `LexEnv`, yielding T_1.
2. Compute the type of $\texttt{exp}_2$ under `LexEnv`, yielding T_2.
3. Constrain T_1 or T_2 to be `String`, or T_1 and T_2 to be numeric but not both `byte`.
4. Coerce $\texttt{exp}_1 : T_1$ and $\texttt{exp}_2 : T_2$ to $\mathrm{add}(T_1, T_2)$.
5. If the expressions at type $\mathrm{add}(T_1, T_2)$ have constant values then let N be the result of applying `+` on them.
   (a) If T_1 and T_2 are singleton numeric literals, return N and $\{N\}$.
   (b) Otherwise, return N and $\mathrm{add}(T_1, T_2)$.
6. Otherwise, return $\mathrm{add}(T_1, T_2)$.

7.6.6.3 Equality Expression

7.6(18) The type of a comparison expression of the form $\texttt{exp}_1$ *op* $\texttt{exp}_2$ where *op* is `==`, `!=`, `===`, or `!==` is computed as follows:

1. Compute the type of $\texttt{exp}_1$ under `LexEnv`, yielding T_1.
2. Compute the type of $\texttt{exp}_2$ under `LexEnv`, yielding T_2.
3. Constrain T_1 and T_2 to be either both numeric types, or both `bool`, or both reference types.
4. Coerce $\texttt{exp}_1 : T_1$ and $\texttt{exp}_2 : T_2$ to $\mathrm{LUB}(T_1, T_2)$.
5. Return `bool`, and if the expressions at type $\mathrm{LUB}(T_1, T_2)$ have constant values then also return the result of applying `==`, `!=`, `===`, or `!==` on them.

7.6.6.4 Shift Expression

7.6(19) The type of a shift or bit arithmetic expression of the form $\texttt{exp}_1$ `<<` $\texttt{exp}_2$ or $\texttt{exp}_1$ `>>` $\texttt{exp}_2$ is computed as follows:

1. Compute the type of $\texttt{exp}_1$ under `LexEnv`, yielding T_1.
2. Compute the type of $\texttt{exp}_2$ under `LexEnv`, yielding T_2.
3. Constrain T_1 to be integral but not `byte`, and coerce $\texttt{exp}_2 : T_2$ to `int`.

4. If the expressions at types T_1 and `int` have constant values then let N be the result of applying `<<` or `>>` on them.
   (a) If T_1 is a singleton numeric literal, return N and $\{N\}$.
   (b) Otherwise, return N and T_1.
5. Otherwise, return T_1.

7.6.6.5 Bit Arithmetic Expression

7.6(20) The type of a shift or bit arithmetic expression of the form $\texttt{exp}_1$ $\texttt{exp}_1$ `&` $\texttt{exp}_2$ or $\texttt{exp}_1$ `|` $\texttt{exp}_2$ or $\texttt{exp}_1$ `^` $\texttt{exp}_2$ is computed as follows:

1. Compute the type of $\texttt{exp}_1$ under `LexEnv`, yielding T_1.
2. Compute the type of $\texttt{exp}_2$ under `LexEnv`, yielding T_2.
3. Constrain T_1 and T_2 to be integral but not both `byte`.
4. Coerce $\texttt{exp}_1 : T_1$ and $\texttt{exp}_2 : T_2$ to $\mathrm{LUB}(T_1, T_2)$.
5. If the expressions at type $\mathrm{LUB}(T_1, T_2)$ have constant values then let N be the result of applying `&`, `|`, or `^` on them.
   (a) If T_1 and T_2 are singleton numeric literals, return N and $\{N\}$.
   (b) Otherwise, return N and $\mathrm{LUB}(T_1, T_2)$.
6. Otherwise, return $\mathrm{LUB}(T_1, T_2)$.

7.6.6.6 Logical Expression

7.6(21) The type of a logical expression of the form $\texttt{exp}_1$ `&&` $\texttt{exp}_2$ or $\texttt{exp}_1$ `||` $\texttt{exp}_2$ is computed as follows:

1. Compute the type of $\texttt{exp}_1$ under `LexEnv`, yielding T_1.
2. Compute the type of $\texttt{exp}_2$ under `LexEnv`, yielding T_2.
3. Coerce $\texttt{exp}_1 : T_1$ and $\texttt{exp}_2 : T_2$ to `bool`.
4. Return `bool`, and if the expressions at type `bool` have constant values then also return the result of applying `&&` or `||` on them.

7.6.6.7 Is Expression

7.6(22) The type of an `is` expression of the form `exp is` T is computed as follows:

1. Compute the type of `exp` under `LexEnv`.
2. Return `bool`, and if the expression has a constant value then also return the result of applying `is` on it.

7.6.6.8 As Expression

7.6(23) The type of an `as` expression of the form `exp as` T is computed as follows:

1. Compute the type of `exp` under `LexEnv`.
2. Return T, and if the expression has a constant value then also return the result of applying `as` on it.

7.6.7 Conditional Expressions

7.6(24) The type of a conditional expression of the form `condexp ? exp`$_1$ `: exp`$_2$ is computed as follows:

1. Compute the type T of `condexp` under `LexEnv`.
2. Coerce `condexp` to `bool`.
3. Compute the type of `exp`$_1$ under `LexEnv`, yielding T_1.
4. Compute the type of `exp`$_2$ under `LexEnv`, yielding T_2.
5. Coerce `exp`$_1$: T_1 and `exp`$_2$: T_2 to LUB(T_1, T_2).
6. If the expressions at type `bool` and LUB(T_1, T_2) have constant values then let N be the result of applying `?...:` on them.
   (a) If T_1 and T_2 are singleton numeric literals, return N and $\{N\}$.
   (b) Otherwise, return N and LUB(T_1, T_2).
7. Otherwise, return LUB(T_1, T_2).

7.6.8 Assignment Statement

7.6(25) An *AssignmentStatement* of the form `r = e` is processed as follows:

1. Let T_1 be the type of `r`.
2. Let T_2 and e be the type and optional constant value of `e`.
3. Coerce `e`: T_2 to T_1, and return.

7.6.9 Return Statement

7.6(26) A *ReturnStatement* is processed as follows:

1. Let T_1 be the return type of the enclosing function definition or function expression.
2. If *ReturnStatement* has an *Expression* e:
   (a) Compute the type and optional constant value of e under `LexEnv`, yielding T_2 and e.
   (b) Coerce $e : T_2$ to T_1.
3. Otherwise, constrain T_1 to be `void`.
4. Return.

7.6.10 Throw Statement

7.6(27) A *ThrowStatement* is processed as follows:

1. Let e be the *Expression* of the *ThrowStatement*.
2. Compute the type and optional constant value of e under `LexEnv`, yielding T and e.
3. Return.

7.6.11 Switch Statement

7.6(28) A *SwitchStatement* is processed as follows:

1. Compute the type and the optional constant value of the *ParenExpression* and each *Expression* in the *CaseClauses* of the *SwitchStatement*.
2. The *Expression*s are treated as if the *ParenExpression* were compared by `==` with each *Expression* in the *CaseClauses*.
3. Return.

7.6.12 Super Statement

7.6(29) A *SuperStatement* is processed as follows:

1. Let the *Expression*s in its *Arguments* be $\mathsf{argexp}_1, \ldots, \mathsf{argexp}_k$.
2. For each $\ell \in \{1, \ldots, k\}$, compute the type and optional constant value of argexp_ℓ under LexEnv, yielding T_ℓ and argexp_ℓ.
3. If the type of the constructor of the base class is $(T'_1, \ldots, T'_i, T'_{i+1}?, \ldots, T'_{i+j}?) \Rightarrow T'$, $i \leq k \leq i+j$, then coerce $\mathsf{argexp}_1 : T_1, \ldots, \mathsf{argexp}_k : T_k$ to $T'_1, \ldots, T'_k$ and return.

7.6.13 Function Definition

7.6(30) A function definition is processed as follows.

1. Let the types of the non-optional *Parameter*s in its *FunctionSignature* be $T_1, \ldots, T_i$, the types and *Expression*s of the *OptionalParameter*s in its *FunctionSignature* be $T_{i+1}, \ldots, T_{i+j}$ and $e_{i+1}, \ldots, e_{i+j}$, and its *ResultType* be T.
2. The expressions $e_{i+1}, \ldots, e_{i+j}$ must have constant values $v_{i+1}, \ldots, v_{i+j}$, and let their types be $T'_{i+1}, \ldots, T'_{i+j}$.
3. Coerce $v_{i+1} : T'_{i+1}, \ldots, v_{i+j} : T'_{i+j}$ to $T_{i+1}, \ldots, T_{i+j}$ and return.

7.7 Type Inference and Constraint Checking

7.7(1) Type inference requires that the programmer specify:

1. parameter types of functions;
2. types of instance vars, static vars

In turn, type inference can recover:

1. return types of functions
2. types of instance lets, static lets
3. types of local variables

7.7(2) The compiler generates coercions between types and constraints on types, as described above. Additionally, some of the coercions are treated as constraints, and do not participate in type inference: a coercion to any type other than a type variable or a `LUB` or `add` type operation is treated as a constraint, which is satisfied if and only if the coercion is valid. When type inference completes, type variables are replaced by their solutions. Thereupon, all constraints must be satisfied, any redundant coercions may be removed, and other coercions are implemented by the *coercion operator* (defined in the next chapter).

7.7(3) Type inference is performed by iterative relabeling of the nodes of a directed graph to a fixpoint. The directed graph has types as nodes, where nodes that involve type variables are initially labeled $\perp$, and all other nodes are labeled by their types. The directed graph has the following two kinds of edges:

1. A *flow edge* between a type and a type variable.
2. An *operation edge* from type operands to type operations over them.

A relaxation of the directed graph consists of the following:

1. For each type variable, an iterated LUB of all labels on incoming types through flow edges is computed, with unit $\perp$, and the type variable is relabeled with that iterated LUB.
2. Type operations whose operands have been relabeled are themselves relabed by recomputing those type operations on those labels.

At fixpoint, the labels of the type variables (which must not be $\perp$) are their solutions.

7.8 Enforcement of `public`, `internal`, `protected`, and `private`

7.8(1) A reference of the form P.id must resolve to a type defined in package P marked `public`.

7.8(2) A reference of the form $o.x$, where the type of o is not `*`, must resolve to a definition *def* whose immediately enclosing scope is the instance scope of a *ClassDefinition* named *name*, such that *name* is the name of the enclosing *ClassDefinition*, or *def* is marked `public`, or *def* is marked `protected` and *name* is the name of a base class of the enclosing *ClassDefinition*, or *def* is marked `internal` and *name* is the name of a *ClassDefinition* in a package with the same name as the package of the enclosing unit.

7.8(3) A reference of the form $C.x$ or **static**.x must resolve to a definition *def* whose immediately enclosing scope is the static scope of a *ClassDefinition* named *name*, and *name* is the name of the enclosing *ClassDefinition*, or *def* is marked `public`, or *def* is marked `protected` and *name* is the name of a base class of the enclosing *ClassDefinition*, or *def* is marked `internal` and *name* is the name of a *ClassDefinition* in a package with the same package name as that the package of the enclosing unit.

7.8(4) A reference of the form `super`.x must resolve to a definition *def* whose immediately enclosing scope is the instance scope of the base class of the enclosing class definition, and it is marked `public` or `protected`, or it is marked `internal` and the reference is enclosed in the same package as the base class.

7.8(5) For an expression of the form `new` $C(e_1, \ldots, e_n)$, C must resolve to a class definition whose constructor is *def*, and either C is the name of the enclosing *ClassDefinition*, or *def* is marked `public`, or *def* is marked `protected` and C is the name of a base class of the enclosing *ClassDefinition*, or *def* is marked `internal` and C is the name of a *ClassDefinition* in a package with the same package name as that the package of the enclosing unit.

7.8(6) For an expression of the form `super` $(e_1, \ldots, e_n)$, let the base class of the enclosing class definition be named C whose constructor is *def*, then *def* is marked `public` or `protected`, or *def* is marked `internal` and C is the name of a *ClassDefinition* in a package with the same package name as that the package of the enclosing unit.

7.9 Enforcement of `let`

7.9(1) The compiler must prove that a `let` is not read before it is written, and that it is written exactly once. At runtime, it is sufficient to disallow writes to a `let` through dynamic code, and have no restriction on its reads through dynamic code.

7.9.1 Local `let`s

7.9(2) The compiler ensures that there is one and only one write to a local `let`, by ensuring that in the control-flow graph of the scope, there is no path from the declaration of the `let` to the end of the scope on which there are zero or multiple *Assignment*s to the `let`. (In particular, functions defined along the path from the declaration of the `let` to the end of the scope must not write to the `let`, because those writes may occur zero or multiple times.)

7.9(3) The compiler ensures that there is no read to a local `let` on any path in the control-flow graph of the scope between the declaration of a `let` and the one and only one write to the `let`. (In particular, functions defined along such a path must not read the `let`, because they may be called.)

7.9.2 Instance `let`s

7.9(4) The compiler ensures that there is one and only one write to an instance `let` as follows:

1. A function other than the constructor must not write to the `let`. Furthermore, a *DeferStatement* in a constructor must not write to the `let`. Finally, the `let` must not be written through an instance member reference outside the scope, i.e., there must not be an assignment to $o.x$ if o is of type T and the definition corresponding to `x` in T is marked `let`. (Note that writes may still be possible in dynamically typed code: e.g., `{dyn:* = this; dyn.x = ...}`.)
2. In the control-flow graph of the constructor, there must not be any path between the super statement and the defer statement on which there are zero or multiple *Assignment*s to the `let`.

7.9(5) The compiler ensures that there is no read to an instance `let` on any path in the control-flow graph of the constructor between the super statement and the one and only one write to the `let`, and that there is no read of `this` other than to write to an instance field, or to read an instance field marked `var`, or read a previously initialized instance field marked `let`, on the path. (Every instance function is assumed to read the `let`.)

7.9.3 Static `let`s

7.9(6) The compiler ensures that there is one and only one write to a static `let` as follows:

1. A function other than the static constructor must not write to the `let`. Furthermore, the `let` must not written through a static member reference outside the scope.
2. In the control-flow graph of the static constructor, there must not be any path from the beginning to the end of the scope on which there is zero or multiple *Assignment*s to the `let`.

7.9(7) The compiler ensures that there is no read to a static `let` on any path in the control-flow graph of the static constructor between the beginning and the one and only one write to the `let`, and there is no **static** reference, or a reference to a static member of a class name, or a reference to a class name in a type expression, other than to write to a static field of the enclosing class, or to read a static field marked `var`, or read a previously initialized static field marked `let` on the path. (Every instance function or static function is assumed to read the `let`.)

7.10 Call Expansion

7.10(1) An *Assignment* of the form $r = e$ (such that r is a *ReferenceExpression*) where r resolves to a setter, is rewritten to the expression $r(e)$.

7.10(2) Any *ReferenceExpression* not involved in an *Assignment*, of the form r, where r resolves to a getter, is rewritten to the expression $r()$.

7.10(3) The default values in the signature of a function definition in a class that overrides another function definition in another class must be the same as in the signature of the overridden function definition.

7.10(4) The default values in the signature of a function definition in a class that implements another function definition in an interface must be the same as in the signature of the implemented function definition.

7.10(5) Any expression of the form $r(e_1, \ldots, e_{i+k})$, where r is a *ReferenceExpression* that resolves to a *FunctionDefinition* whose type is $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow T$, where $k \leq j$, is rewritten to the expression $r(e_1, \ldots, e_{i+k}, v_{i+k+1}, \ldots, v_{i+j})$, where $v_{i+1}, \ldots, v_{i+j}$ are the default values in the signature of the *FunctionDefinition*.

7.10(6) Any expression of the form `new` $C(e_1, \ldots, e_{i+k})$, where C resolves to a class definition whose constructor's type is $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow$`void`, where $k \leq j$, is rewritten to the expression `new` $C(e_1, \ldots, e_{i+k}, v_{i+k+1}, \ldots, v_{i+j})$, where $v_{i+1}, \ldots, v_{i+j}$ are the default values in the signature of the constructor.

7.10(7) Any expression of the form `super` $(e_1, \ldots, e_{i+k})$, where the type of the constructor of the base class of the enclosing definition is $(T_1, \ldots, T_i, T_{i+1}?, \ldots, T_{i+j}?) \Rightarrow$`void`, where $k \leq j$, is rewritten to the expression `super` $(e_1, \ldots, e_{i+k}, v_{i+k+1}, \ldots, v_{i+j})$, where $v_{i+1}, \ldots, v_{i+j}$ are the default values in the signature of the constructor.

Part IV

Execution

Chapter 8

Run-Time Structures and Operations

This chapter describes how execution proceeds against *run-time environments* that contains run-time representations of global scopes (consisting of types) and local scopes. In particular, it describes how run-time environments are *allocated*, and specifies data structures and the semantics of various intrinsic operations that manipulate those data structures at run time, while interacting with run-time environments.

8.1 Run-Time Environments

8.1(1) Run-time environments are used to store and look up representations that correspond to the information contained in lexical environments.

Definition 8.1.1 (Location)**.** A location is a block of memory of known type, that is *uninitialized* when allocated, and otherwise always contains some value of that type.

Definition 8.1.2 (Run-Time Environment)**.** A run-time environment is a set of run-time bindings. A run-time binding is a pair of a name and a location.

8.1(2) There is a global run-time environment that maps names of types (classes and interfaces) to their representations during execution.

8.2 Values and Types

Definition 8.2.1 (Run-Time Type)**.** A *run-time type* is either a numeric or boolean type, or a function closure type, or a class.

Definition 8.2.2 (Class)**.** A class carries a *ClassInheritance*, a *ClassBody*, and a run-time environment.

Definition 8.2.3 (Value)**.** A *value* is either a numeric or boolean value or an *object*.

Definition 8.2.4 (Object)**.** Every object carries a run-time type and a run-time environment.

Definition 8.2.5 (Box)**.** A box is an object carrying a numeric or boolean value.

Definition 8.2.6 (Function closure)**.** A function closure is an object carrying a run-time environment, and whose type carries a *FunctionBody*.

Definition 8.2.7 (Array)**.** An array is an object carrying a (fixed) length and a sequence of locations of that length.

Definition 8.2.8 (ArrayList)**.** An array list is an object carrying a (variable) length and a sequence of locations of that length.

Definition 8.2.9 (String)**.** A string is an object carrying a (fixed) length and a sequence of characters of that length in the Unicode encoding.

8.3 Instrinsic Operations

8.3(1) To simplify the presentation of dynamic evaluation semantics, code is transformed to the following intrinsic operations, as described in the next chapter.

8.3.1 GetLocal

8.3(2) The operation **GetLocal**(x) finds the location mapped to x in the run-time environment of the enclosing function closure, gets the value contained in the location and returns it.

8.3.2 SetLocal

8.3(3) The operation **SetLocal**($x = e$) finds the location mapped to x in the run-time environment of the enclosing function closure, evaluates e to a value v, and puts v into the location.

8.3.3 GetStaticLex

8.3(4) The operation **GetStaticLex**(**static**.x) finds the location mapped to x in the run-time environment of the enclosing class, gets the value contained in the location, and returns it.

8.3.4 SetStaticLex

8.3(5) The operation **SetStaticLex**(**static**.$x = e$) finds the location mapped to x in the run-time environment of the enclosing class, evaluates e to a value v, and puts v into the location.

8.3.5 GetStaticMember

8.3(6) The operation **GetStaticMember**($C.x$) finds the location mapped to x in the run-time environment of C, gets the value contained in the location, and returns it.

8.3.6 SetStaticMember

8.3(7) The operation **SetStaticMember**($C.x = e$) finds the location mapped to x in the run-time environment of C, evaluates e to a value v, and puts v into the location.

8.3.7 GetInstanceField

8.3(8) The operation **GetInstanceField**($e_0.x$) evaluates e to an object o, finds the location mapped to x in run-time environment of o, gets the value contained in the location, and returns it.

8.3.8 SetInstanceField

8.3(9) The operation **SetInstanceField**($e_0.x = e$) evaluates e_0 to an object o, finds the location mapped to x in run-time environment of o, evaluates e to the value v, and puts v into the location.

8.3.9 GetInstanceMethod

8.3(10) The operation **GetInstanceMethod**($e_0.x$) evaluates e_0 to an object o, performs **GetType** on o to obtain C, then performs **GetStaticMember**$C.x$ to obtain f, and finally performs **Call**($f(o)$).

8.3.10 GetSuperMethod

8.3(11) The operation **GetSuperMethod**(`super`.x) performs **GetStaticMember**$C.x$ where C is the super class of the enclosing class to obtain f, finally performs **Call**(f(`this`)).

8.3.11 $*$::GetInstanceMember

8.3(12) The operation ***::GetInstanceMember**($e_0.x$) evaluates e_0 to an object o. Next, it performs **GetType** on o to obtain C, and looks up x in the instance scope of C yielding a definition *def* of type T. Enforce access control on the definition: *def*'s immediately enclosing scope must be the instance scope of a *ClassDefinition* named *name*, such that *name* is the name of the enclosing *ClassDefinition*, or *def* is marked `public`, or *def* is marked `protected` and *name* is the name of a base class of the enclosing *ClassDefinition*, or *def* is marked `internal` and *name* is the name of a *ClassDefinition* in a package with the same name as the package of the enclosing unit.

1. If the definition is a variable definition, perform **GetInstanceField**($o.x$)
2. If the definition is a getter, perform **Call**(**GetInstanceMethod**($o.x$)()).
3. Otherwise the definition is any other function definition, perform **GetInstanceMethod**($o.x$).

Perform T**::Op**(v `coerce` $*$) on the result v, and return it.

8.3.12 $*$::SetInstanceMember

8.3(13) The operation ***::SetInstanceMember**($e_0.x = e$) evaluates e_0 to an object o. Next, it performs **GetType** on o to obtain C, and looks up x in the instance scope of C yielding a definition *def*. Enforce the access control on the definition: *def*'s immediately enclosing scope must be the instance scope of a *ClassDefinition* named *name*, such that *name* is the name of the enclosing *ClassDefinition*, or *def* is marked `public`, or *def* is marked `protected` and *name* is the name of a base class of the enclosing *ClassDefinition*, or *def* is marked `internal` and *name* is the name of a *ClassDefinition* in a package with the same name as the package of the enclosing unit. Also, e is evaluated to an object v.

1. If the definition is a variable definition, then it must not be marked `let`, let T be its type. Perform ***::Op**(v `coerce` T) to obtain v', and perform **SetInstanceField**($o.x = v'$).
2. Otherwise the definition must be a setter, let T be its type. Perform ***::Op**(v `coerce` T) to obtain v', and then perform **Call**(**GetInstanceMethod**($o.x$)(v')).

8.3.13 Call

8.3(14) The operation **Call**$(e_0(e_1, \ldots, e_k))$ evaluates e_0 to a function closure f and $e_1, \ldots, e_k$ to values $v_1, \ldots, v_k$.

1. Allocation is performed for the invocation of f.
2. The locations corresponding to the parameters in the run-time environment built during allocation are initialized with $v_1, \ldots, v_k$.
3. The function body of f is run, switching to its run-time environment.
4. Any return jump or error is caught.
5. The run-time environment is switched back to the current run-time environment.
6. The return value, if it exists, is returned, or the error, if it exists, is re-thrown.

8.3.14 $*$::CallReturn

8.3(15) The operation **Call**$(e_0(e_1, \ldots, e_{i+k}))$ evaluates e_0 to a function closure f. Next, it looks up the function signature of f, whose non-optional parameter types are $T_1, \ldots, T_i$, optional parameter types are $T_{i+1}, \ldots, T_{i+j}$ with default values $v''_{i+1}, \ldots, v''_{i+j}$, and result type is T. Next, it evaluates $e_1, \ldots, e_{i+k}$ to objects $v_1, \ldots, v_{i+k}$, and performs $*$**::Op**$(v_m$ `coerce` $T_m)$ for each $m \in 1..i+k$ to obtain values $v'_1, \ldots, v'_{i+k}$. Finally, it performs **Call**$(f(v'_1, \ldots, v'_{i+k}, v''_{i+k+1}, \ldots, v''_{i+j}))$, performs T**::Op**$(v$ `coerce` $*)$ on the result, and returns it.

8.3.15 $*$::Call

8.3(16) The operation **Call**$(e_0(e_1, \ldots, e_{i+k}))$ evaluates e_0 to a function closure f. Next, it looks up the function signature of f, whose non-optional parameter types are $T_1, \ldots, T_i$, optional parameter types are $T_{i+1}, \ldots, T_{i+j}$ with default values $v''_{i+1}, \ldots, v''_{i+j}$. Next, it evaluates $e_1, \ldots, e_{i+k}$ to objects $v_1, \ldots, v_{i+k}$, and performs $*$**::Op**$(v_m$ `coerce` $T_m)$ for each $m \in 1..i+k$ to obtain values $v'_1, \ldots, v'_{i+k}$. Finally, it performs **Call**$(f(v'_1, \ldots, v'_{i+k}, v''_{i+k+1}, \ldots, v''_{i+j}))$.

8.3.16 NewInstance

8.3(17) The operation **NewInstance**(`new` $C(e_1, \ldots, e_k)$) evaluates $e_1, \ldots, e_k$ to values $v_1, \ldots, v_k$, creates a new object o of type C, performs allocation for o, performs **GetInstanceMethod**(o.`%init%`) to obtain the constructor f, and finally performs **Call**$(f(v_1, \ldots, v_k)())$ and returns o.

8.3.17 NewFunction

8.3(18) The operation **NewFunction**(F) where F is a function expression makes a new function closure whose function body is derived from the function, and whose run-time environment is the current run-time environment (which is empty in a global scope).

8.3.18 NewArray

8.3(19) The operation **NewArray**(`new[]`$T\{e_1, \ldots, e_k\}$) evaluates $e_1, \ldots, e_k$ to values $v_1, \ldots, v_k$, and creates a new array whose length is k and whose sequence of locations are initialized with $v_1, \ldots, v_k$.

8.3.19 GetType

8.3(20) The operation **GetType**(o) evaluates o to an object and returns it type.

8.3.20 Unbox

8.3(21) The operation **Unbox**(o) evaluates o to a box, and returns the boolean or numeric value carried by the box.

8.3.21 <type>::GetIndex

8.3(22) The operation T**::GetIndex**($a[i]$) evaluates a to an object a' and i to an int i'.

1. Either T is an array type or array list type. Assert that i' is between 0 and $n-1$ where n is a'`.length`, find the location at that index, and get the value contained in the location.
2. Or T is `*`. Perform **GetType**(a') to obtain T'. Assert that T' is an array type or array list type whose element type is T'', perform T'**::GetIndex**($a'[i']$) to obtain v', perform T''**::Op**(v' `coerce` $*$) and return it.

8.3.22 <type>::SetIndex

8.3(23) The operation T**::SetIndex**($a[i] = e$) evaluates a to an object a', i to an int i', and e to a value v.

1. Either T is an array type or array list type. Assert that i' is between 0 and $n-1$ where n is a'`.length`, find the location at that index, and put v into the location.
2. Or T is `*`. Perform **GetType**(a') to obtain T'. Assert that T' is an array type or array list type whose element type is T'', perform $*$ :: **Op**(v `coerce` T'') to obtain v', and perform T'**::SetIndex**($a'[i'] = v'$).

8.3.23 <type>::Op

8.3(24) The operation T**::Op**(`~`e) evaluates e to a value v.

1. Either T is integral but not byte. Perform `~` (bitwise not) on $v : T$ and return the result of type T.
2. Or T is `*`. Perform **GetType**(v) to obtain T'. Assert that T' is integral but not byte, perform **Unbox**(v) to obtain v', perform T'**::Op**(`~`v') to obtain r, perform T'**::Op**(r `coerce` $*$) and return it.

8.3(25) The operation `bool` :: **Op**(`!`e) evaluates e to a value v. Perform `!` (logical not) on v : `bool` and return the result of type `bool`.

8.3(26) The operation T**::Op**(e_1 *op* e_2), where *op* is `*`, `/`, `%`, `-`, `<`, `<=`, `>`, or `>=`, evaluates e_1 and e_2 to values v_1 and v_2.

1. Either T is numeric but not byte. Perform *op* (multiplicative operation, subtraction, or relational operation) on $v_1 : T$ and $v_2 : T$ and return the result of type T.
2. Or T is `*`. Perform **GetType**(v_1) and **GetType**(v_2) to obtain T_1' and T_2'. Compute $\mathsf{LUB}(T_1', T_2')$ to obtain T', assert that T' is numeric but not byte, perform **Unbox**(v_1) and **Unbox**(v_2) to obtain v_1' and v_2', perform T_1'**::Op**(v_1' `coerce` T') and T_2'**::Op**(v_2' `coerce` T') to obtain v_1'' and v_2'', perform T'**::Op**(v_1'' *op* v_2'') to obtain r, perform T'**::Op**(r `coerce` $*$) and return it.

8.3(27) The operation T**::Op**(e_1`+`e_2) evaluates e_1 and e_2 to values v_1 and v_2.

1. Either T is numeric but not byte. Perform `+` (addition) on $v_1 : T$ and $v_2 : T$ and return the result of type T.

2. Or T is `String`. Perform v_1`.concat`(v_2) and return the result of type `String`.

3. Or T is `*`. Perform **GetType**(v_1) and **GetType**(v_2) to obtain T_1' and T_2'. Compute $\mathsf{add}(T_1', T_2')$ to obtain T'. Assert that T' is either numeric but not byte, in which case perform **Unbox**(v_1) and **Unbox**(v_2) to obtain v_1' and v_2', or `String`, in which case let v_1' and v_2' be v_1 and v_2. Perform T_1'**::Op**$(v_1'$ `coerce` $T')$ and T_2'**::Op**$(v_2'$ `coerce` $T')$ to obtain v_1'' and v_2'', perform T'**::Op**$(v_1''$`+`$v_2'')$ to obtain r, perform T'**::Op**$(r$ `coerce` $*)$ and return it.

8.3(28) The operation T**::Op**$(e_1\ op\ e_2)$, where op is `==` or `!=`, evaluates e_1 and e_2 to values v_1 and v_2.

1. Either T is a value type. Perform op (structural equality or inequality) on $v_1 : T$ and $v_2 : T$ and return the result of type `bool`.

2. Or T is a subtype of `Object`. Perform v_1`.equal`(v_2) or $!v_1$`.equal`(v_2) and return the result of type `bool`.

3. Or T is `*`. Perform **GetType**(v_1) and **GetType**(v_2) to obtain T_1' and T_2'. Compute $\mathsf{LUB}(T_1', T_2')$ to obtain T'. Assert that T' is not `*`, and perform **Unbox**(v_1) and **Unbox**(v_2) to obtain v_1' and v_2'. Perform T_1'**::Op**$(v_1'$ `coerce` $T')$ and T_2'**::Op**$(v_2'$ `coerce` $T')$ to obtain v_1'' and v_2'', perform T'**::Op**$(v_1''\ op\ v_2'')$ and return the result.

8.3(29) The operation T**::Op**$(e_1\ op\ e_2)$, where op is `===` or `!==`, evaluates e_1 and e_2 to values v_1 and v_2.

1. Either T is a value type. Perform op (physical equality or inequality) on $v_1 : T$ and $v_2 : T$ and return the result of type `bool`.

2. Or T is a subtype of `Object`. Perform v_1`.identical`(v_2) or $!v_1$`.identical`(v_2) and return the result of type `bool`.

3. Or T is `*`. Perform **GetType**(v_1) and **GetType**(v_2) to obtain T_1' and T_2'. Compute $\mathsf{LUB}(T_1', T_2')$ to obtain T'. Assert that T' is not `*`, and perform **Unbox**(v_1) and **Unbox**(v_2) to obtain v_1' and v_2'. Perform T_1'**::Op**$(v_1'$ `coerce` $T')$ and T_2'**::Op**$(v_2'$ `coerce` $T')$ to obtain v_1'' and v_2'', perform T'**::Op**$(v_1''\ op\ v_2'')$ and return the result.

8.3(30) The operation T**::Op**$(e_1\ op\ e_2)$, where op is `<<`, `>>`, `&`, `|`, or `^`, evaluates e_1 and e_2 to values v_1 and v_2.

1. Either T is integral but not byte. Perform op (shift operation or bitwise arithmetic operation) on $v_1 : T$ and $v_2 : T$ and return the result of type T.

2. Or T is `*`. Perform **GetType**(v_1) and **GetType**(v_2) to obtain T_1' and T_2'. Compute $\mathsf{LUB}(T_1', T_2')$ to obtain T'. Assert that T' is integral but not byte. Perform **Unbox**(v_1) and **Unbox**(v_2) to obtain v_1' and v_2'. Perform T_1'**::Op**$(v_1'$ `coerce` $T')$ and T_2'**::Op**$(v_2'$ `coerce` $T')$ to obtain v_1'' and v_2'', perform T'**::Op**$(v_1''\ op\ v_2'')$ to obtain r, perform T'**::Op**$(r$ `coerce` $*)$ and return it.

8.3(31) The operation `bool` :: **Op**$(e_1\ op\ e_2)$, where op is `&&` or `||`, evaluates e_1 and e_2 to values v_1 and v_2. Perform op (logical conjuction or disjunction) on $v_1 : T$ and $v_2 : T$ and return the result.

8.3(32) The operation T'**::Op**$(e$ `is` $T)$ evaluates e to a value v.

1. If T' is a value type, then return whether T is the same as T'.

2. Otherwise, perform **GetType**(v) to obtain T'', and return whether T'' is a subtype of T.

8.3(33) The operation T'**::Op**$(e$ `as` $T)$ evaluates e to a value v.

1. If T is `*`, return v if it is an object, otherwise return a box carrying the (numeric or boolean) value with type T.

2. If T' is a value type,

(a) If T is a value type, convert $v : T'$ to T, and return the result.

(b) If T is `String` convert $v : T'$ to `String`, and return the result.

3. Otherwise, perform **GetType**(v) to obtain T''.

   (a) If T'' is a value type, perform **Unbox**(v) to obtain v', perform T''**::Op**(v' `as` T), and return the result.

   (b) If T'' is a subclass of T, return v.

   (c) Otherwise, assert that T is `String`, perform v.`toString`() and return the result.

8.3(34) The operation T'**::Op**(e `coerce` T) evaluates e to a value v.

1. If T is `*`, return v if it is an object, otherwise return a box carrying the (numeric or boolean) value with type T.
2. If T' is a value type and $v : T'$ is promotable or implicitly convertible to T, perform the promotion or implicit conversion and return the result.
3. Otherwise, perform **GetType**(v) to obtain T''.

   (a) If T'' is a value type, perform **Unbox**(v) to obtain v', perform T''**::Op**(v' `coerce` T), and return the result.

   (b) Otherwise, assert that T'' is a subclass of T, and return v.

8.4 Allocation

8.4(1) Allocation is the process of building run-time environments.

Definition 8.4.1 (Default value)**.** The default value for a type is as follows:

1. If the type is `*` or any other reference type, the default value is `null`.
2. If the type is numeric, the default value is the zero of that numeric type.
3. If the type is bool, the default value is false.

8.4(2) Allocation of a class C proceeds as follows:

1. A run-time environment is built, pairing names to locations for the (instance and) static methods and static variables of the class, and copying the non-conflicting run-time bindings of the base class.
2. Those locations corresponding to the (instance and) static methods defined in this class are initialized by corresponding function closures constructed by **NewFunction** with empty run-time environments.
3. Those locations that correspond to static variables marked `var` defined in this class are initialized with their default values based on their types.

8.4(3) Allocation of an instance of class C proceeds as follows:

1. A run-time environment is built, pairing names to locations for the instance variables of the class and its super classes.
2. Those locations that correspond to instance variables marked `var` are initialized with their default values based on their types.

8.4(4) Allocation of a function invocation of function F proceeds as follows:

1. A run-time environment is built, pairing names to locations for the parameters, local variables, and local functions of the function, and copying the run-time bindings of the enclosing local scope, if any.
2. Those locations that correspond to local functions defined in the function are initialized with their corresponding function closures constructed by **NewFunction** with the current run-time environment.
3. Those locations that correspond to local variables marked `var` defined in the function are initialized with their default values based on their types.

8.4(5) Allocation for a block proceeds as follows:

1. A run-time environment is built, pairing names to locations for the local variables and local functions of the function, and copying the run-time bindings of the enclosing local scope, if any.
2. Those locations that correspond to local functions defined in the function are initialized with their corresponding function closures constructed by **NewFunction** with the current run-time environment.
3. Those locations that correspond to local variables marked `var` defined in the function are initialized with their default values based on their types.

Chapter 9

Linking, Verification, and Evaluation

This chapter describes the processes of linking and initialization of classes that serve to regenerate lexical envionments and trigger execution of an executable. It also describes the processes of verification and execution of statements and expressions; in particular, verification involves the transformation of expressions to intrinsic operations, whose evaluation semantics are defined in the previous chapter.

9.1 Linking and Initialization of Classes

9.1(1) Execution of an executable is triggered by the *initialization* of the class that contains the static method that serves as the entry point, followed by *calling* the static method itself. (The entry point's type is asserted to be `() => void`.)

9.1(2) A class must be *linked* before it is initialized.

9.1(3) The linking of a class or interface proceeds as follows:

1. The base class and the base interfaces must be linked (if they have not already been linked).
2. The class or interface is visited to build lexical environments, as prescribed for compilation, and the definition is added to the global lexical environment.

9.1(4) Linking a type must not recursively trigger linking of the same type: this indicates that the type recursively extends itself, which results in an error.

9.1(5) Calling a function must be preceded by the verification of the body of that function (if it has not already been verified).

9.1(6) Any types that appear in some code, and are thus required for verification of that code, must be linked before verification proceeds.

9.1(7) Initialization of a class C proceeds as follows:

1. The base class must be initialized (if it has not already been initialized).
2. A constructor of the form

```
function C(params) {
  super(args);
  ...;
  defer stmt;
```

```
}
```

is translated to an instance method of the form

```
function %init%(params) {
  let k:()=>void = super.%init%(args);
  ...;
  return function() { k(); stmt; }; }
}
```

3. Any instance method of the form

   ```
   function f(params) {...}
   ```

   is translated to a static method of the form

   ```
   static function f(this:C) { return function(params) {...}; }
   ```

4. Allocation for the class is performed.

5. A class is built that carries the *ClassInheritance* and the *ClassBody*, as well as the maps built by allocation, and is added to the global run-time environment.

6. The static constructor of the class is found by **GetStaticMember**(C.**static**), yielding a function closure f that is called by **Call**($f()$).

9.1(8) Initialization of a class must not recursively trigger initialization of the same class: this indicates that the class recursively depends on itself, which results in an error.

9.2 Verification

9.2(1) Dynamic verification closely mimics static verification, except that there is no promotability or implicit coercibility. In addition to type constraints, enforcement of access control and `let` semantics is carried out as prescribed for compilation.

9.2.1 References

9.2(2) A local read of the form x is verified and translated as follows:

1. Look up x in the lexical environment, yielding a definition; let its type be T_1.
2. Translate the expression to **GetLocal**(x) and return its type as T_1.

9.2(3) A local write of the form $x = e$ is verified and translated as follows:

1. Look up x in the lexical environment, yielding a definition; let the type of the variable definition be T_1.
2. Verify and translate e to e', let its type be T_2.
3. Assert that T_2 is a subtype of T_1.
4. Translate the expression to **SetLocal**($x = e'$).

9.2(4) An instance member read of the form $o.x$ is verified and translated as follows:

1. Verify and translate o to o', let its type be T.
2. If T is `*`, translate the expression to `*`::**GetInstanceMember**($o'.x$) and return its type as `*`.

3. Otherwise, look up `x` in the lexical environment associated with type T, yielding a definition. Let T_1 be the type of the definition. If the definition is of a variable, translate the expression as the operation **GetInstanceField**$(o'.x)$; otherwise the definition is of a function, translate the expression as **GetInstanceMethod**$(o'.x)$. Return its type as T_1.

9.2(5) An instance member write of the form $o.x = e$ is verified and translated as follows:

1. Verify and translate o to o', let its type be T.
2. Verify and translate e to e', let its type be T_2.
3. If T is `*`, assert that T_2 is `*`, and translate the expression to `*`**::SetInstanceMember**$(o'.x = e')$.
4. Otherwise, look up `x` in the lexical environment associated with type T, yielding a variable definition. Let T_1 be the type of the definition. Assert that T_2 is a subtype of T_1 and the definition is of a variable, translate the expression as **SetInstanceField**$(o'.x = e')$.

9.2(6) A super method read of the form `super`.x is verified and translated as follows:

1. Look up x in the lexical environment of the instance scope of the base class of the enclosing class definition, yielding a function definition.
2. Let T be the type of the definition.
3. Translate the expression as **GetSuperMethod**(`super`.x) and return its type as T.

9.2(7) A static member read of the form **static**.x is verified and translated as follows:

1. Look up x in the lexical environment of the static scope of the enclosing class definition, yielding a definition.
2. Let T_1 be its type.
3. Translate the expression as **GetStaticLex**(**static**.x) and return T_1 as its type.

9.2(8) A static field write of the form **static**.$x = e$ is verified and translated as follows:

1. Look up x in the lexical environment of the static scope of the enclosing class definition, yielding a variable definition.
2. Let T_1 be its type.
3. Verify and translate e to e', let its type be T_2.
4. Assert that T_2 is a subtype of T_1.
5. Translate the expression as **SetStaticLex**(**static**.$x = e'$).

9.2(9) A static member read of the form $C.x$ is verified and translated as follows:

1. Look up x in the lexical environment of the static scope of the class definition C, yielding a definition.
2. Assert that the definition appears in the class body of C.
3. Let T_1 be its type.
4. Translate the expression as **GetStaticMember**$(C.x)$ and return T_1 as its type.

9.2(10) A static field write of the form $C.x = e$ is verified and translated as follows:

1. Look up x in the lexical environment of the static scope of the class definition C, yielding a variable definition.
2. Assert that the definition appears in the class body of C.
3. Let T_1 be its type.

4. Verify and translate e to e', let its type be T_2.
5. Assert that T_2 is a subtype of T_1.
6. Translate the expression as $\mathbf{SetStaticMember}(C.x = e')$.

9.2(11) An element read of the form `a[i]` is verified and translated as follows:

1. Verify and translate a to a', let its type be T.
2. Verify and translate i to i', and assert that its type is `int`.
3. If T is `*` then translate the expression as `*`$\mathbf{::GetIndex}(a'[i'])$ and return its type as `*`.
4. Otherwise, assert that T is `[]`T_1 or `ArrayList<`T_1`>`, translate the expression as $T\mathbf{::GetIndex}(a'[i'])$, and return its type as T_1.

9.2(12) An element write of the form `a[i] = e` is verified and translated as follows:

1. Verify and translate a to a', let its type be T.
2. Verify and translate i to i', and assert that its type is `int`.
3. Verify and translate e to e', let its type be T_2.
4. If T is `*` then assert that T_2 is `*` and translate the expression as `*`$\mathbf{::SetIndex}(a'[i'] = e')$.
5. Otherwise, assert that T is `[]`T_1 or `ArrayList<`T_1`>`, and that T_2 is a subtype of T, and translate the expression as $T\mathbf{::SetIndex}(a'[i'] = e')$.

9.2.2 Literals

9.2.2.1 Null Literal

9.2(13) Evaluate `null` to the value `null` of type {`null`}.

9.2.2.2 Boolean Literals

9.2(14) Evaluate `true` or `false` to the corresponding value of type `bool`.

9.2.2.3 Number Literals

9.2(15) Evaluate an integral *NumericLiteral* to a value of type `int` (32-bit signed integer).

9.2(16) Evaluate a non-integral *NumericLiteral* to a value of type `double` (double precision floating point number as defined in IEEE 754).

9.2.2.4 String Literals

9.2(17) Evaluate a *StringLiteral* to a string.

9.2.3 Primary Expressions

9.2.3.1 Regular Expression Initializers

9.2(18) Verify and translate a *regular expression initializer* of the form /pattern/flags, as follows:

1. Assert that pattern and flags are strings.
2. Translate the expression to `new RegExp(pattern,flags)` of type `RegExp`.

9.2.3.2 Array Initializers

9.2(19) An *array initializer* of the form $\texttt{new}[]T\{\mathsf{expr}_1, \ldots, \mathsf{expr}_k\}$, where T is a type and $\mathsf{expr}_1, \ldots, \mathsf{expr}_k$ is a sequence of expressions, is verified and translated as follows:

1. Verify and translate the sequence of expressions $\mathsf{expr}_1, \ldots, \mathsf{expr}_k$ to $\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k$.
2. Assert that the types of $\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k$ are subtypes of `T`.
3. Translate the expression to **NewArray**($\texttt{new}[]T\{\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k\}$) of type `[]T`.

9.2.3.3 Function Initializers

9.2(20) A *FunctionInitializer* is verified and translated as follows:

1. Translate the expression to **NewFunction**(*FunctionBody*).
2. Return the function type derived from the *FunctionSignature*.

9.2.3.4 This

9.2(21) Verify and translate `this` as follows:

1. Assert that it is enclosed by the instance scope of some class C.
2. Translate the expression as **GetLocal**(`this`) of type C.

9.2.4 Call Expression

9.2(22) Verify and translate a *call expression* of the form $\mathsf{expr}_0(\mathsf{expr}_1, ..., \mathsf{expr}_k)$, where expr_0 is an expression and $\mathsf{expr}_1, ..., \mathsf{expr}_k$ is a sequence of expressions, as follows:

1. Verify and translate expr_0 to f of type T.
2. Verify and translate $\mathsf{expr}_1, \ldots, \mathsf{expr}_k$ to $\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k$ of types $T'_1, \ldots, T'_k$.
3. If T is `*`, then assert that $T'_1, \ldots, T_k$ are `*`. If the expression is not a statement, translate the expression to ***::CallReturn**($f(\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k)$); otherwise translate the expression to ***::Call**($f(\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k)$). Return `*`.
4. Otherwise T is a function type $(T_1, \ldots, T_k) \Rightarrow T'$. Assert that $T'_1, \ldots, T_k$ are subtypes of $T_1, \ldots, T_k$. Translate the expression to **Call**($f(\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k)$) and return T'.

9.2.5 New Expression

9.2(23) Verify and translate a *new expression* of the form `new` C`(`$\mathsf{expr}_1, ..., \mathsf{expr}_k$`)`, where expr_0 is an expression and $\mathsf{expr}_1, ..., \mathsf{expr}_k$ is a sequence of expressions, as follows:

1. Let the function type $(T_1, \ldots, T_k) \Rightarrow T'$ be the type of the constructor of C.
2. Verify and translate $\mathsf{expr}_1, \ldots, \mathsf{expr}_k$ to $\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k$ of types $T'_1, \ldots, T'_k$.
3. Assert that $T'_1, \ldots, T_k$ are subtypes of $T_1, \ldots, T_k$. Translate the expression to **NewInstance**(`new` C $(\mathsf{expr}'_1, \ldots, \mathsf{expr}'_k)$) and return C.

9.2.6 Unary Expressions

9.2.6.1 Bitwise Not Expression

9.2(24) An expression of the form `~argexp` is verified and translated as follows:

1. Verify and translate `argexp` to e, let its type be T.
2. Assert that T is either integral but not `byte` or `*`.
3. Translate the expression to T**::Op**(`~`e), and return T.

9.2.6.2 Logical Not Expression

9.2(25) An expression of the form `!argexp` is verified and translated as follows:

1. Verify and translate `argexp` to e, let its type be T.
2. Assert that T is `bool`.
3. Translate the expression to `bool`**::Op**(`!`e), and return `bool`.

9.2.7 Binary Expressions

9.2.7.1 Multiplicative Expression, Subtract Expression, Relational Expression

9.2(26) An expression of the form exp_1 *op* exp_2, where *op* is `*`, `/`, `%`, `-`, `<`, `<=`, `>`, or `>=` is verified and translated as follows:

1. Verify and translate exp_1 and exp_2 to e_1 and e_2, let their types be T.
2. Assert that T is either numeric but not `byte`, or `*`.
3. Translate the expression to T**::Op**(e_1 *op* e_2) and return T.

9.2.7.2 Add Expression

9.2(27) An expression of the form exp_1 `+` exp_2 is verified and translated as follows:

1. Verify and translate exp_1 and exp_2 to e_1 and e_2, let their types be T.
2. Assert that T is either numeric but not `byte`, or `String`, or `*`.
3. Translate the expression to T**::Op**($e_1 + e_2$) and return T.

9.2.7.3 Equality Expression

9.2(28) An expression of the form $\texttt{exp}_1$ *op* $\texttt{exp}_2$, where *op* is `==`, `!=`, `===`, or `!==`, is verified and translated as follows:

1. Verify and translate $\texttt{exp}_1$ and $\texttt{exp}_2$ to e_1 and e_2, let their types be T.
2. Translate the expression to T**::Op**$(e_1\ op\ e_2)$ and return `bool`.

9.2.7.4 Shift Expression, Bit Arithmetic Expression

9.2(29) An expression of the form $\texttt{exp}_1$ *op* $\texttt{exp}_2$, where *op* is `<<`, `>>`, `&`, `|`, or `^`, is verified and translated as follows:

1. Verify and translate $\texttt{exp}_1$ and $\texttt{exp}_2$ to e_1 and e_2, let their types be T.
2. Assert that T is either integral but not `byte`, or `*`.
3. Translate the expression to T**::Op**$(e_1\ op\ e_2)$ and return T.

9.2.7.5 Logical Expression

9.2(30) An expression of the form $\texttt{exp}_1$ *op* $\texttt{exp}_2$, where *op* is `&&` or `||`, is verified and translated as follows:

1. Verify and translate $\texttt{exp}_1$ and $\texttt{exp}_2$ to e_1 and e_2, let their types be T.
2. Assert that T is `bool`.
3. Translate the expression to `bool`**::Op**$(e_1\ op\ e_2)$ and return `bool`.

9.2.7.6 Is Expression

9.2(31) An expression of the form `exp is` T is verified and translated as follows:

1. Verify and translate `exp` to e, let its type be T'.
2. Translate the expression to T'**::Op**$(e$ `is` $T)$ and return `bool`.

9.2.7.7 As Expression

9.2(32) An expression of the form `exp as` T is verified and translated as follows:

1. Verify and translate `exp` to e, let its type be T'.
2. Translate the expression to T'**::Op**$(e$ `as` $T)$ and return T.

9.2.8 Statements

9.2.8.1 If Statements

9.2(33) An *IfStatement* is verified by ensuring that the type of its "condition" *Expression* is `bool`.

9.2.8.2 Switch Statements

9.2(34) A *SwitchStatement* is verified by ensuring that the types of the *ParenExpression* and each *Expression* in the *CaseClauses* is the same.

9.2.8.3 While Statements

9.2(35) An *WhileStatement* is verified by ensuring that the type of its "condition" *Expression* is `bool`.

9.2.8.4 Return Statements

9.2(36) A *ReturnStatement* is verified by ensuring that the type of its *Expression* matches the return type of the enclosing function.

9.3 Execution

9.3.1 Labeled Statements

Definition 9.3.1 (Next iteration)**.** The next iteration of an iteration statement is defined as follows:

1. If the iteration statement is a *LabeledStatement*, return the next iteration of the *Statement* of the *LabeledStatement*.
2. Otherwise, return the iteration statement.

9.3(1) A *LabeledStatement* is executed as follows:

1. Try executing its *Statement*.
2. If execution completes, return.
3. Otherwise:
   (a) If a "break" jump is thrown whose label is that of the *LabeledStatement*, return.
   (b) If a "continue" jump is thrown whose label is that of the *LabeledStatement*, execute the next iteration of the *Statement* of the *LabeledStatement*.

9.3.2 Block Statements

9.3(2) A *BlockStatement* is executed as follows:

1. Allocation is performed for the scope.
2. Execute each *Statement* in turn.

9.3.3 If Statements

9.3(3) An *IfStatement* is executed as follows:

1. Evaluate the "condition" *Expression*, yielding a boolean value.

2. If the value is `true`, execute the "then" *Statement*.
3. Otherwise, the value is `false`; execute the "else" *Statement*.

9.3.4 Switch Statements

9.3(4) A *SwitchStatement* is executed as follows:

1. Evaluate the "switch" *Expression*, yielding a value V.
2. Let *matched* be false.
3. For each *CaseClause* in *CaseClauses*:
   (a) If *matched* is false, evaluate the "case" *Expression*, yielding a value. If the value compares by `==` to V, let *matched* be true.
   (b) If *matched* is true, try executing the "case" *Statement*; if a "break" jump is thrown without a label, return.
4. If there is a *DefaultClause*, try executing the "case" *Statement*; if a "break" jump is thrown without a label, return.

9.3.5 While Statements

9.3(5) A *WhileStatement* is executed as follows:

1. Repeat:
   (a) Evaluate the "condition" *Expression*, yielding a boolean value.
   (b) If false, return.
   (c) Otherwise, try executing the "body" *Statement*.
   (d) If a "break" jump is thrown without a label, return.
   (e) if a "continue" jump is thrown without a label, skip.

9.3.6 Break Statements

9.3(6) A *BreakStatement* is executed as follows:

1. Throw a "break" jump, with the *Label* of the *BreakStatement* if it exists.

9.3.7 Continue Statements

9.3(7) A *ContinueStatement* is executed as follows:

1. Throw a "continue" jump, with the *Label* of the *ContinueStatement* if it exists.

9.3.7.1 Return Statements

9.3(8) A *ReturnStatement* is executed as follows:

1. If the *ReturnStatement* has an *Expression*, evaluate it to a value; throw a "return" jump with the value.
2. Otherwise, throw a "return" jump.

9.3.8 Throw Statements

9.3(9) A *ThrowStatement* is executed as follows:

1. Evaluate the *Expression* of the *ThrowStatement*, yielding a value.
2. Throw an exception with the value.

9.3.9 Try Statements

1. Try executing the "try" *Block*.
2. If there is no exception, execute the "finally" *Block*.
3. Otherwise, let there be an exception with value V. For each *CatchClause* in *CatchClauses*, if V is coercible via `is` to the type of the *TypedBinding*, then try executing the *Block*.
   (a) If there is an exception with value V, execute the "finally" *Block* and throw V.
   (b) Otherwise, execute the "finally" *Block* and return.
4. Throw V.