The ActionScript 3 Language Specification
Constant Evaluation

Avik Chaudhuri, Jeff Dyer, Lars Hansen, and Basil Hosmer

Adobe Systems Inc.
`{achaudhu,jodyer,lhansen,bhosmer}@adobe.com`

October 28, 2011

© 2011 Adobe Systems Incorporated and its licensors. All rights reserved.

Adobe ActionScript Language 3 Specification Version 1.0

This user guide is protected under copyright law, furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

This guide contains links to third-party websites that are not under the control of Adobe Systems Incorporated, and Adobe Systems Incorporated is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Adobe Systems Incorporated provides these links only as a convenience, and the inclusion of the link does not imply that Adobe Systems Incorporated endorses or accepts any responsibility for the content on those third-party sites. No right, license, or interest is granted in any third party technology referenced in this guide.

This user guide is licensed for use under the terms of the Creative Commons Attribution Non-Commercial 3.0 License. This License allows users to copy, distribute, and transmit the user guide for noncommercial purposes only so long as (1) proper attribution to Adobe is given as the owner of the user guide; and (2) any reuse or distribution of the user guide contains a notice that use of the user guide is governed by these terms. The best way to provide notice is to include the following link. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/.

Adobe, ActionScript, Flash, and Flash Player are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. All other trademarks are the property of their respective owners.

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

Notice to U.S. Government End Users: The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

1 Overview

1(1) Constant evaluation is the process of evaluating expressions at compile time, after *lexical environment building* and *definition hoisting* and before *static verification*. The effect of constant evaluation is that some expressions are replaced by their constant values, and some lexical bindings are given the constant values of their initializer expressions and therefore have those values upon run time variable instantiation.

1(2) Constant expressions are required in some contexts, affect the meaning of phrases in other contexts, and may be simplified to the literal equivalent to their constant value in all other contexts. In the case that constant evaluation does not change the meaning of an expression, the simplification of that expression is optional. In particular:

1. An expression that occurs in an *OptionalParameter* to denote the default value of that parameter must be a constant expression.
2. A constant expression that occurs in the syntactic nonterminal *VariableInitialization* with the `const` keyword causes the corresponding **var** slot to have the value of that constant expression, thus making it a *constant binding*.
3. In a *class context*, a constant expression that occurs in the syntactic nonterminal *VariableInitialization* with the `var` keyword causes the corresponding **var** slot to have the initial value of that constant expression.
4. A constant expression with the static type **Number** that occurs where a float value is expected (as in an assignment to a float typed location), or where a float value would change the meaning of an operation (as in a binary expression involving a float value), is inferred to be a float value if it is an integer in the contiguous range of integers that can be exactly represented in IEEE 754 binary floating point encoding (-2^{24} to 2^{24}, inclusive.) (See the spec "A 'float' Data Type of AS3" for full details.)

1(3) The lexical environment used for constant evaluation includes the bindings of the global environment tables of any imported *Program*s. In particular this means that before constant evaluation begins, all imported bindings are inserted into the global environment table and therefore participate in constant evaluation.

2 Imported Bindings and Global Environments

Definition 2.1 (Exported binding)**.** An *exported binding* is a binding defined in one program that is available for use in other programs. All of the bindings in the global environment table of a program are *exported bindings* regardless of their visibility.

Definition 2.2 (Imported binding)**.** An *imported binding* is an exported binding that is made available to a program by being added to its global environment table. Imported bindings are added when the global environment table is initialized (see Names and Lexical Environments, section 5.3.2).

2(1) The mechanisms by which bindings are caused to be imported are outside of the language but one common means is a compiler switch (e.g. ASC -import) that causes all of the the exported bindings of one program to be imported to the program being compiled.

2(2) Imported bindings participate in constant evaluation in the same way as other bindings. In particular, references to imported class bindings yield environment tables that may contain constant bindings, and references to imported constant bindings yield constant values.

2(3) The value of an imported class or constant binding is computed before it is imported and therefore is not affected by constant evaluation of the current program.

2(4) It is an error for an environment table to contain two bindings with the same name (this is not always the case with ASC, *ASL-286*), therefore the order in which bindings are imported is of no significance to constant evaluation.

2(5) One consequence of imported bindings participating in constant evaluation is that the same set of bindings may not be available at compile time and run time potentially changing the meaning of names between the two phases. This is an issue shared between other modern programming languages including *Java*, *C#* and *C++*.

3 Evaluating Expressions

3(1) Constant evaluation of an expression under a lexical environment yields either a value or an *unknown value* result. In particular, constant evaluation of any expression not covered below returns an *unknown value* result.

3.1 Property Name Expressions

3(2) Constant evaluation of expressions relies on constant simplification of property names.

Definition 3.1 (Constant simplification of a property name expression)**.** The *constant simplification of a property name expression* under a lexical environment LexEnv yields either a *multiname* or an *unknown name* result, as follows:

```
1. If the property name expression is a multiname $mname$, return $mname$.
2. If the property name expression is a late multiname of the form $\{nsval_1, \ldots, nsval_k\}$ :: $[expr]$:
   (a) Evaluate $expr$ under LexEnv, yielding the result $R$.
   (b) If $R$ is a value $val$,
       i. Call ToString($val$) yielding $id$.
       ii. Return $\{nsval_1, \ldots, nsval_k\}$::$id$.
```

3. If the property name expression is of the form *nsexpr*::*id*:
   (a) Evaluate *nsexpr* under LexEnv, yielding the result *R*.
   (b) If *R* is a namespace value *nsval*, return {*nsval*}::*id*.
4. If the property name expression is of the form *nsexpr*:: [*expr*]:
   (a) Evaluate *nsexpr* under LexEnv, yielding a result *R*.
   (b) Evaluate *expr* under LexEnv, yielding a result R'.
   (c) If *R* is a namespace value *nsval* and R' is a value *val*,
      i. Call **ToString**(*val*) yielding *id*.
      ii. Return {*nsval*}::*id*.
5. Otherwise, return the *unknown name* result.

3.2 Lexical Reference Expressions

Definition 3.2 (Constant binding)**.** A *constant binding* is a lexical binding whose slot is of either of the following forms:

1. **namespace**(...)
2. **var**(...) that is marked `const` and carries a value

3(3) Constant evaluation of a lexical reference expression of the form *propexpr*, where *propexpr* is a property name expression, under a lexical environment LexEnv, proceeds as follows:

1. Perform *constant simplification* of propexpr under LexEnv, yielding a result *R*.
2. If *R* is a multiname *mname*:
   (a) Call *lookup*(*mname*, LexEnv), yielding a result R'. (See Definition 4.5 (Multiname lookup) in "Names and Lexical Envrionments")
   (b) If R' is a constant binding, return the initializer value of the lexical binding.
3. Otherwise, return the *unknown value* result.

3(4) **Example.** Constant evaluation of lexical references. In the following example, all three variable definitions, *c*1 through *c*3, have initializers that are lexical references with property name expressions that simplify to a multiname that refers to a constant binding, and therefore they all result in constant bindings themselves.

```
namespace ns;
const str = "x"
class A {
    ns static const x = 10;
    use namespace ns;
    const c1 = x;
    const c2 = ns::x;
    const c3 = ns::[str];
}
```

3.3 Object Reference Expressions

3(5) Constant evaluation of an object reference expression of the form mname.propexpr, where mname is a multiname and propexpr is a property name expression, under a lexical environment LexEnv, proceeds as follows:

1. Call *lookup*(mname, LexEnv), yielding a result R.
2. If R is a lexical binding and the slot of the lexical binding is of the form **class**(...):
   (a) Let table be the static environment table in the **class**(...) slot.
   (b) Perform *constant simplification* of propexpr under LexEnv, yielding a result R'.
   (c) If R' is a multiname *mname*$'$:
      i. Call *lookup*(*mname*$'$, [table]), yielding a result R''.
      ii. If R'' is a constant binding, return the initializer value of the constant binding.
3. Otherwise, return the *unknown value* result.

3(6) **Example.** Constant evaluation of object references. In the following example, all four variable definitions, $c1$ through $c4$, have initializers that are object references with property name expressions that simplify to a multiname that refers to a constant binding, the therefore they all result in constant bindings themselves.

```
namespace ns;
const str = "x"
class A {
    ns static const x = 10;
}
use namespace ns;
const c1 = A.x;
const c2 = A[str];
const c3 = A.ns::x;
const c4 = A.ns::[str];
```

3.4 Literal Expressions

3(7) Constant evaluation of a StringLiteral, a NumericLiteral, `null`, `true`, or `false`, under a lexical environment, return whatever result would be yielded by the run time evaluation of the expression.

3.5 Operator Expressions

3(8) Constant evaluation of an expression that involves the operator `typeof`, `void`, `!`, `||`, `&&`, `!=`, `==`, `!==`, `===`, `~`, `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `>`, `>=`, `<<`, `>>`, `>>>`, `&`, `|`, `^`, `is`, `as`, `?...:...`, or `(...)` under a lexical environment LexEnv, proceeds as follows:

1. Evaluate the subexpression(s) under LexEnv, yielding results R (and R').
2. If R (and R') are values, return the result that would be yielded by the run time evaluation of the operator on those values.
3. Otherwise, return the *unknown value* result.

3.6 Function Call and New Expressions

Definition 3.3 (Value type)**.** A value type is one of the builtin classes in the namespace **public**(`""`): `int`, `uint`, `Number`, `String`, `Boolean`, `float`, and `float4`.

3(9) Constant evaluation of a function call expression of the form $mname(expr_1, \ldots, expr_k)$ and `new` $mname(expr_1, \ldots, expr_k)$, where *mname* is a multiname and $expr_1, \ldots, expr_k$ is a sequence of expressions, under a lexical environment LexEnv, proceeds as follows:

1. Call *lookup*(mname, LexEnv), yielding a result R.
2. If R is a lexical binding whose QName is a value type T:
   (a) Evaluate $expr_1, \ldots, expr_k$ under LexEnv, yielding a sequence of results $R_1, \ldots, R_k$.
   (b) If $R_1, \ldots, R_k$ is a sequence of values $val_1, \ldots, val_k$, return the value that would be yielded by a call expression of T applied to $val_1, \ldots, val_k$.
3. Otherwise, return the *unknown value* result.

3(10) **Example.** Constant evaluation of call expressions. In the following example, all three const variable definitions, $c1$ through $c3$, have initializers that are constant call expressions and therefore result in constant bindings themselves.

```
const c1 = String(10);
const c2 = int("10");
const c3 = float4(1,2,3,4);
```

4 Evaluating Programs and Updating Lexical Bindings

4(1) Constant evaluation proceeds over a program in a depth first manner with lexical bindings being updated as their definitions' initializers are found to be constant expressions. Recall that *definition hoisting* has already moved some definitions in outer blocks to locations before the definitions of constant bindings in inner blocks (See the chapter titled "Definition Hoisting", to be written).

Definition 4.1 (Constant evaluation in a defining scope)**.** Let the lexical environment LexEnv be associated with a defining scope. The process of *constant evaluation* in that scope affects definitions with constant initializers as follows:

1. For an *OptionalParameter* whose *Type* is T:
   (a) Evaluate the *NonAssignmentExpression* of the *OptionalParameter* under LexEnv, yielding a result R.
   (b) If R is a value *val*, replace the *NonAssignmentExpression* of the *OptionalParameter* with the result of coercing *val* to T.
   (c) Otherwise, report an error.
2. For a *VariableDefinition* immediately nested in a *Block* that corresponds to a defining scope and whose *VariableKind* is `const` and *Type* is T: (Note: the requirement of immediate nesting in a defining scope fixes ASL-287)
   (a) Evaluate the *VariableInitializer* under LexEnv, yielding a result R.

   (b) If R is a value *val*, update the lexical binding associated with the *VariableDefinition* in the environment table of the defining scope by modifying it to be a constant binding with the value that results from coercing *val* to T.

3. For a *VariableDefinition* in a *class context* whose *VariableKind* is `var` and *Type* is T: (Note: this rule codifies existing AS3 behavior. ASL-288 proposes to fix the special handling of class variables by removing this rule.)
   (a) Evaluate the *VariableInitializer* under LexEnv, yielding a result R.
   (b) If R is a value *val*, update the lexical binding associated with the *VariableDefinition* in the environment table of the defining scope so that it carries the value that results from coercing *val* to T.

4. For every other expression in LexEnv:
   (a) Evaluate the expression under LexEnv, yielding a result R.
   (b) If R is a value, replace the expression with the literal expression that denotes that value.

4(2) **Example.** The following example shows some constant and non-constant bindings that result from different initializers.

```
class A {
    print(x, y, z, w); // prints: undefined undefined 30 30
    static const x = y;         // non-constant binding, forward reference to non-constant binding
    static const y = z + 30;  // non-constant binding, forward reference to constant binding
    static const z = 10 + 20; // constant binding
    static const w = z;         // constant binding
    print(x, y, z, w); // prints: undefined 60 30 30
}
```

4(3) **Example.** The appearence of sorting of initializers can occur when definitions have initializers that are constant and so result in bindings that have values and get initialized upon variable instantiation.

```
class A {
    print(x, y); // prints: 10 undefined
    static const y = x; // gets value of 10 at run time
    static var x = 10; // gets value of 10 at compile time
    print(x, y); // prints: 10 10
}
```

Example. The following example shows that const variables are not constant bindings when declared in a nested block.

```
if (true) {
    const c = 0;
    function f(x=c) {} // error, c not a constant
}
```

5 Appendix: Compatibility with AS3

5(1) The constant evaluation semantics defined in this chapter is signifcantly stronger than what has been implemented in ASC to date. While the benefits of constant evaluation are signficant and virtually without cost to new programs, there are some subtle changes in behavior to existing programs that are recompiled with semantics defined here. These include:

1. Changes in initialization order
2. Changes in the resolution time of references
3. Changes in value of certain floating point expressions
4. Changes in errors from duplicate external bindings
5. Changes to handling of const variables in nested blocks

Changes in initialization order

5(2) In AS3, if a variable initializer is a constant expression the value of that expression is used to initialize the variable when the variable is instantiated rather than when the variable definition is encountered at runtime. With stronger constant evaluation, more initializers become constant expressions potentially causing the order of initialization of variables to change in ways that might be difficult to anticipate.

5(3) Also, this spec proposes to change the handling of variable initializers with constant expressions, which will change the meaning of programs by not ever initializing such variables early. (See ASL-288).

Changes in the resolution time of references

5(4) With separate compilation the set of external bindings might change between compile time and run time. With stronger constant evaluation names that were previously resolved at run time might now be resolved at compile time. If the set of bindings in scope of such a reference changes between compile time and run time, for example because of dynamic loading, then the value yielded by certain references could change.

Changes in the value of floating point expressions

5(5) The AVM has a non-standard rounding algorithm for floating pointing numbers, which is difficult and generally undesireable to propagate to constant evaluation which implements the standard algorithm. The consequence is that expressions involving floating point numbers which used to be evaluated by at run time and are now evaluated at compile time may change value at the highest digits of precision.

Changes in errors from duplicate external bindings

5(6) ASC allows duplicate imported variable bindings, even if they are constant bindings with different types and values. The specification outlaws such duplicates, which will cause some programs that used to compile to have errors.

Changes to the handling of const variables in nested blocks

5(7) ASC considers a 'const' definition with a constant initializer to be a constant binding regardless of where it occurs. This spec proposes to require such bindings to be outside of a Block to be considered a constant binding. This avoids confusion due to differing hoisting rules for different kinds of definitions. (See ASL-287).

6 Appendix: Definition Hoisting

6(1) Recall that *NamespaceDefinition*s have already been moved into the lexical environment of its defining scope during *lexical environment building* (See "Names and Lexical Environments for the details.) Definition hoisting, which follows *lexical environment building* and precedes *constant evaluation*, results in the moving of some remaining definitions from their original location to an earlier location in the syntax tree. This phasing is important because it means that affected definitions in outer blocks will be moved before constants in inner blocks and therefore references from those inner definitions may become constant as a result.

6(2) During definition hoisting, a program is transformed as follows:

1. *ClassDefinition*s, *InterfaceDefinition*s, *FunctionDefinition*s and *VariableDefinition*s marked by the `const` keyword are moved in the order that they appear according to the following rules:
   (a) In a *SwitchStatement* context, they are inserted at the beginning of the *Directives* of the current *CaseClause*.
   (b) In a *Block* context, they are inserted at the beginning of the *Directives* of the current *Block*.
   (c) In a *Program* context, they are inserted at the beginning of the *Directives* of the current *Program*.

6(3)

6(4) **Example.** The following example shows the effect of definition hoisting. The lexical reference to 'c' from the *OptionalParameter* in 'f' is a constant expression because the definition of 'c' is moved to the top of the program before constant evaluation.

```
if (true) {
    function f(x=c) { } // okay, 'c' is a constant and is hoisted to the top of its block
}
else {
}
const c = 10;
```

7 Appendix: Changes assumed to have been made to the Names chapter

7(1) This appendix enumerates changes that are assumed to be made to the May 16, 2011 draft of Names.

Modification of slots

7(2) We extend the following kinds of slots:

1. **var**(. . .) slots optionally carry a value (the initializer value of a variable, if such exists)
2. **class** slots carry two environment tables (the instance environment table and the static environment table of a class).

Retrieving the value of a slot

7(3) We define the following general procedure for computing the initializer value of a lexical binding.

Definition 7.1 (Initializer value of a lexical binding)**.** The *initializer value of a lexical binding* is defined as follows:

1. If the lexical binding is of the form $(\_, \mathbf{namespace}(\mathsf{nsval}))$, return nsval.
2. If the lexical binding is of the form $(\_, \mathbf{var}(\ldots))$ and the slot carries a value, return that value.
3. Otherwise, return the *unknown value* result.

Resolution of expanded names

7(4) An expanded name is resolved by a lexical environment to a lexical binding. Resolution always ends in the lookup of a multiname.