
Alfa Reference Manual

Release 0.1

AdaCore

December 09, 2011

CONTENTS

1	General	1
1.1	Purity	1
1.2	Dynamic Semantics	1
2	Lexical Elements	3
3	Declarations and Types	5
3.1	Declarations	5
3.2	Types and Subtypes	5
3.3	Objects and Named Numbers	5
3.4	Derived Types and Classes	5
3.5	Scalar Types	5
3.6	Array Types	5
3.7	Discriminants	6
3.8	Record Types	6
3.9	Tagged Types and Type Extensions	6
3.10	Access Types	6
3.11	Declarative Parts	6
4	Names and Expressions	7
4.1	Names	7
4.2	Literals	7
4.3	Aggregates	7
4.4	Expressions	7
4.5	Operators and Expression Evaluation	7
4.6	Type Conversions	7
4.7	Qualified Expressions	8
4.8	Allocators	8
4.9	Static Expressions and Static Subtypes	8
5	Statements	9
5.1	Simple and Compound Statements - Sequences of Statements	9
5.2	Assignment Statements	9
5.3	If Statements	9
5.4	Case Statements	9
5.5	Loop Statements	9
5.6	Block Statements	9
5.7	Exit Statements	9
5.8	Goto Statements	10

6	Subprograms	11
6.1	Subprogram Declarations	11
6.2	Formal Parameter Modes	12
6.3	Subprogram Bodies	12
6.4	Subprogram Calls	12
6.5	Return Statements	12
6.6	Overloading of Operators	12
6.7	Null Procedures	13
6.8	Expression Functions	13
7	Packages	15
7.1	Private Types and Private Extensions	15
7.2	Type Invariants	15
7.3	Assignment and Finalization	15
8	Visibility Rules	17
8.1	Use Clauses	17
8.2	Renaming Declarations	17
9	Tasks and Synchronization	19
10	Program Structure and Verification Issues	21
11	Exceptions	23
12	Generic Units	25
13	Representation Issues	27
13.1	Representation Clauses	27
13.2	Machine Code Insertions	27
13.3	Unchecked Type Conversions	27
13.4	Storage Management	27
13.5	Streams	27
13.6	Interfacing to Other Languages	27
14	GNAT Specificities	29
14.1	Implementation-Defined Pragmas	29

GENERAL

Alfa is a subset of Ada 2012 suitable for automatic formal verification of programs. Alfa builds on the capability of specifying contracts for subprograms provided in Ada 2012. Alfa supports modular verification of subprograms by unit testing or unit proving: a subprogram with a contract can be unit tested; a subprogram in Alfa with a contract can also be unit proved. In order to combine the results of unit testing and unit proving, the complete program should be *Alfa friendly*, so that the assumptions made during unit proving of a subprogram can be dynamically verified during unit testing of a caller or callee of this subprogram.

Alfa restricts language features to remove the possibility of nondeterminism and to make automatic proof possible. For example, it excludes access types, exceptions, and controlled types, and it requires functions (but not procedures) to be pure. Some restrictions are syntactic (e.g., `explicit_dereference` is not in Alfa) and other restrictions are semantic (e.g., `implicit_dereference` is not in Alfa). Unless stated otherwise, a construct is in Alfa.

The Alfa-friendly profile restricts language features so that the global parameters of subprograms are computable, and aliasing can be detected. For example, it excludes controlled types and calls through access-to-subprogram values. It is equivalent to the following set of restrictions:

```
pragma Restrictions (  
    No_Access_Subprograms,  
    No_Finalization,  
    No_Implicit_Aliasing,  
    No_Parameter_Aliasing,  
    No_Uninitialized_Parameters);
```

Note that some restrictions can be detected statically, while others (aliasing, initialization) must be detected at run-time. This document defines both the Alfa subset of Ada 2012 and the Alfa-friendly profile.

1.1 Purity

In the rest of this document, we say that a construct is pure, or, equivalently, free from side effects, if its evaluation cannot modify the value of a variable or memory location. Note that this is not the same as the Ada term “pure”, which applies to packages whose declarations satisfy certain criteria that are somewhat related to but not the same as the Alfa notion of purity.

1.2 Dynamic Semantics

The Alfa-friendly profile augments the dynamic semantics of Ada with additional run-time checks. At the cost of this additional verification, we get a valid combination between unit testing and unit proving of subprograms in an Alfa-friendly program. To get this benefit, tools for unit testing/proving must take into account the additional run-time checks. For example, this can be a special mode of the compiler for unit testing.

LEXICAL ELEMENTS

The following reserved words are not in Alfa: **abort**, **accept**, **access**, **aliased**, **delay**, **entry**, **exception**, **goto**, **protected**, **raise**, **requeue**, **select**, **synchronized**, **task**, **terminate**, **until**.

DECLARATIONS AND TYPES

3.1 Declarations

The view of an entity is in Alfa if and only if the corresponding declaration is in Alfa. When clear from the context, we say *entity* instead of using the more formal term *view of an entity*.

3.2 Types and Subtypes

The view of an entity introduced by a `private_type_declaration` is in Alfa, even if the entity declared by the corresponding `full_type_declaration` is not in Alfa.

For a type or subtype to be in Alfa, all predicate specifications that apply to the (sub)type must be in Alfa. Notwithstanding any rule to the contrary, a (sub)type is never in Alfa if its applicable predicate is not in Alfa.

3.3 Objects and Named Numbers

No specific restrictions. Thus, the entity declared by an object declaration is in Alfa if its declaration does not contain the reserved word `aliased`, its type is in Alfa, and its initialization expression, if any, is in Alfa.

3.4 Derived Types and Classes

No specific restrictions.

3.5 Scalar Types

No specific restrictions.

3.6 Array Types

No specific restrictions.

3.7 Discriminants

No specific restrictions.

3.8 Record Types

No specific restrictions.

3.9 Tagged Types and Type Extensions

No specific restrictions.

3.10 Access Types

Access types allow the creation of aliased data structures and objects, which notably complicate the specification and verification of a program's behavior. This is the reason access types are excluded from Alfa. This falls out without any specific restrictions from the reserved word `access` not being in Alfa.

Additionally, values of type `access-to-subprogram` make it impossible to compute the global parameters of subprograms and to detect aliasing. This excludes all forms of `access-to-subprogram` types in Alfa friendly code.

3.11 Declarative Parts

No specific restrictions.

NAMES AND EXPRESSIONS

4.1 Names

A name that denotes an entity is in Alfa if and only if the entity is in Alfa. Neither `explicit_dereference` nor `implicit_dereference` are in Alfa.

Attribute `Access` is not in Alfa. As they are based on access discriminants, user-defined references and user-defined indexing are not in Alfa.

4.2 Literals

The literal `null` is not in Alfa.

4.3 Aggregates

An aggregate is in Alfa only if its type is in Alfa and it is pure.

4.4 Expressions

An expression is in Alfa only if its type is in Alfa and it is pure.

4.5 Operators and Expression Evaluation

In Alfa, there can be no sequence of operators of the same precedence level. Parentheses must be used to impose specific associations.

4.6 Type Conversions

No specific restrictions.

4.7 Qualified Expressions

No specific restrictions.

4.8 Allocators

Allocators are not in Alfa.

4.9 Static Expressions and Static Subtypes

No specific restrictions.

STATEMENTS

5.1 Simple and Compound Statements - Sequences of Statements

No specific restrictions.

5.2 Assignment Statements

No specific restrictions.

5.3 If Statements

No specific restrictions.

5.4 Case Statements

No specific restrictions.

5.5 Loop Statements

User-defined iterator types are not in Alfa. An `iterator_specification` is not in Alfa.

5.6 Block Statements

No specific restrictions.

5.7 Exit Statements

No specific restrictions.

5.8 Goto Statements

Goto statements are not in Alfa.

SUBPROGRAMS

We distinguish the *declaration view* introduced by a `subprogram_declaration` from the *implementation view* introduced by a `subprogram_body` or an `expression_function_declaration`. For subprograms that are not declared by a `subprogram_declaration`, the `subprogram_body` or `expression_function_declaration` also introduces a declaration view which may be in Alfa even if the implementation view is not.

6.1 Subprogram Declarations

A function is in Alfa only if it is pure.

6.1.1 Preconditions and Postconditions

As indicated by the `aspect_specification` being part of a `subprogram_declaration`, a subprogram is in Alfa only if its specific contract expressions (introduced by `Pre` and `Post`) and class-wide contract expressions (introduced by `Pre'Class` and `Post'Class`), if any, are in Alfa.

6.1.2 Global Parameters

Besides reading and writing its parameters, a subprogram in Ada may directly reference objects in scope at the point of the reference, as well as locations pointed to through dereferences of a non-local access type. Note that such references may occur directly in the body of the subprogram, or in the body of another called subprogram. Objects other than formal parameters that are referenced either directly by their name or via dereferences of a non-local access type are referred to as the *global parameters* of a subprogram.

GNAT provides a way to specify the global parameters of a subprogram:

```
Global [in|out|in out]? => Annotation_List
Annotation_List ::= ( Annotation_Item { Annotation_Item} )
Annotation_Item ::= object_name | NULL
```

Item `object_name` shall identify an object in scope, while **null**, if present, shall be the only item in the list. Specifying `Global => null` on an imported function states that the subprogram does not have global parameters. An imported function is in Alfa only if it has an annotation giving its global parameters.

Here are examples of use of global parameters:

```
procedure Get_Obj with Global in => Obj;
procedure Set_Obj with Global out => Obj;
procedure Incr_Obj with Global in out => Obj;
procedure Copy with Global in => From, Global out => To;
procedure Mult_Copy with Global in => From1 From2, Global out => To1 To2;
function Pure_Func (X, Y : T) return Boolean with Global => null;
```

6.2 Formal Parameter Modes

In Alfa-friendly code, it is not allowed in a call to pass as parameters references to overlapping locations, when at least one of the parameters is of mode `out` or `in out`, unless the other parameter is of mode `in` and `by-copy`. Likewise, it is not allowed in a call to pass as `out` or `in out` parameter a reference to some location which overlaps with any global parameter of the subprogram. Finally, it is not allowed in a call to pass as `in` or `in out` parameter a reference to some location which overlaps with a global parameter of mode `out` or `in out` of the subprogram, unless the parameter is of mode `in` and `by-copy`. This is the meaning of restriction `No_Parameter_Aliasing`.

A parameter `X` of type `T` is said to be valid if:

- `X'Valid` evaluates to `True` for a scalar type `T`;
- all components are valid for a record type `T`, where the validity of a variant component is defined as the validity of the components for its current value of discriminant;
- all indexed components are valid for an array type `T`.

In Alfa-friendly code, all `in` and `in out` parameters should be valid at the point of call. And all `out` and `in out` parameters should be valid when returning from the subprogram. This is the meaning of restriction `No_Uninitialized_Parameters`.

6.3 Subprogram Bodies

No specific restrictions.

6.4 Subprogram Calls

A call is in Alfa only if it resolves statically to a subprogram whose declaration view is in Alfa (whether the call is dispatching or not).

6.5 Return Statements

No specific restrictions.

6.6 Overloading of Operators

No specific restrictions.

6.7 Null Procedures

No specific restrictions.

6.8 Expression Functions

No specific restrictions.

PACKAGES

Packages themselves are not classified as being Alfa or non-Alfa. Rather, declarations within a package are individually classified as to whether they are in Alfa.

7.1 Private Types and Private Extensions

The partial view of a private type or private extension may be in Alfa even if its full view is not in Alfa. The usual rule applies here, so a private type without discriminants is in Alfa, while a private type with discriminants is in Alfa only if its discriminants are in Alfa.

7.2 Type Invariants

The Ada 2012 RM lists places at which an invariant check is performed. In Alfa, we add the following places:

- Before a call on any subprogram or entry that:
 - is explicitly declared within the immediate scope of type T (or by an instance of a generic unit, and the generic is declared within the immediate scope of type T), and
 - is visible outside the immediate scope of type T or overrides an operation that is visible outside the immediate scope of T, and
 - has one or more in out or in parameters with a part of type T.

the check is performed on each such part of type T.

7.3 Assignment and Finalization

Controlled types are not Alfa friendly.

VISIBILITY RULES

8.1 Use Clauses

Use clauses are always in Alfa.

8.2 Renaming Declarations

No specific restrictions.

TASKS AND SYNCHRONIZATION

Concurrent programs require the use of different specification and verification techniques from sequential programs. This is the reason why concurrency constructs are excluded from Alfa.

PROGRAM STRUCTURE AND VERIFICATION ISSUES

Because functions and expressions in Alfa are pure, it is necessary to analyze the bodies of all subprograms called to determine if a function or an expression is in Alfa.

EXCEPTIONS

Raising and handling of exceptions allow forms of control flow that complicate both specification and verification of a program's behavior. This is the reason explicit uses of exceptions are excluded from Alfa.

Note that the reserved word *exception* is not in Alfa, thus handlers are not in Alfa either.

Exceptions can still be raised implicitly (for example, by the failure of a language-defined check). Pragmas `Assert`, `Assertion_Policy`, `Suppress`, and `Unsuppress` are in Alfa.

GENERIC UNITS

Generic units are not classified as being Alfa or non-Alfa. Only their instantiations are.

REPRESENTATION ISSUES

13.1 Representation Clauses

Representation clauses are in Alfa, except for `'Address` representation clauses.

13.2 Machine Code Insertions

Machine code insertions are not in Alfa.

13.3 Unchecked Type Conversions

Unchecked type conversions are not in Alfa.

13.4 Storage Management

Storage pools are not Alfa friendly, like other controlled types.

13.5 Streams

`Stream Read` and `Write` operations are not in Alfa.

13.6 Interfacing to Other Languages

`Pragma` or aspect `Unchecked_Union` is not in Alfa.

GNAT SPECIFICITIES

14.1 Implementation-Defined Pragmas

Pragma `Debug` is accepted in Alfa, and it has no effect.