ASIS vs. LibAdalang: a Comparative Assesment

J-P. Rosen
Adalog, 2 rue du Docteur Lombard, 92130 Issy-Les-Moulineaux, FRANCE; email: rosen@adalog.fr

Abstract

This paper compares the origins, features, and status of two different tools intended to facilitate static analysis of Ada programs: ASIS and LibAdalang.

It stresses the differences in principles, features, and intended usages, and shows use cases where each is more appropriate.

Keywords:Ada, static analysis, ASIS, LibAdalang.

1 Introduction

Ada is a language which is both large and complex to compile. Any tool that aims at providing some form of static analysis of an Ada program must cope with visibility rules, overloading resolution, generic instantiations, defaulted parameters, etc.

For example, consider the simple statement:

```
V := A (B);
```

Possible interpretations are:

- A is a function, B is the parameter
- A is an array, B is an index
- A is a type, B is converted to A
- A is a parameterless function call returning an array, B is an index
- A is a pointer to a function, B is the parameter

And of course, B can be a constant, a variable, or a function call...

Starting an analysis tool from scratch would require almost rewriting half of a compiler. To avoid this effort and foster the development of language tools, ASIS was developed as an API to access the decorated AST (*Abstract Syntax Tree*) produced by the compiler. More recently, AdaCore [2] developed an alternative solution named LibAdalang.

Developers of Ada tools have now two competing solutions, and are faced with fundamental decisions:

- When developing a new tool, which solution is more appropriate?
- When evolving an ASIS tool, is it appropriate to invest time and money for moving it to LibAdalang?

This paper aims at providing some material to help answering these questions.

2 Origin

ASIS

ASIS [1] is an international standard, first developed for Ada 83, then updated to Ada 95. It was developed by an international committee (the ASIS working group), and built upon experience gained from previous attempts to standardize an intermediate representation for Ada, notably DIANA [3].

Several compilers provide the ASIS interface. The standard was not evolved for subsequent updates of Ada, however implementations, especially the AdaCore one, continued to add support up to Ada 2012. AdaCore announced however that their implementation would not be upgraded to support Ada 2022.

LibAdalang

LibAdalang is an independent, self-funded development of AdaCore. It is developed by a dedicated team from AdaCore. It is an open-source project[1] available on GitHub [2], and the team welcomes comments from the community; however, there is no control of any official or international body over the design decisions.

3 Fundamental principles

ASIS

ASIS is an API to explore and get information from the decorated syntactic tree, as produced by the associated compiler. This guarantees that a tool based on ASIS will see the code exactly as the compiler sees it, including implementation dependent elements allowed by the standard, and elements defined in the System and Standard packages. However, this implies that an ASIS implementation is linked to a certain compiler. A tool based on ASIS tool must provide specific versions for each supported compiler.

Since ASIS operates on a tree resulting from a successful compilation, it cannot handle incorrect or incomplete code. For the same reason, it was a deliberate design decision to *not* provide any operation that would modify, or even add information, to the syntactic tree. It is purely oriented towards analysing a program, with no way to modify it.

LibAdalang

LibAdalang includes its own analyser and tree constructor, which is part of the library, and embedded with any

[1] This is assumed from the long-time involvement of AdaCore with free software. However, at the time of writing, the project distribution has no mention saying “LibAdalang is free software”; this is likely to be an omission, but it makes the copyright status unclear.

application that uses it. A LibAdalang application is therefore stand-alone, independent of any compiler, and can be used even without an Ada compiler on the target machine. On the other hand, there is no guarantee that the view of the program provided to the tool is strictly equivalent to the compiler's view, nor that packages System and Standard match the ones of the compiled code. More precisely, LibAdalang will happily accept any program which is *syntactically* correct, even if it is not *semantically* correct. For example, no error is diagnosed in the following program:

```
procedure Incorrect is
  I : Integer;
begin
  I := 1.0;  -- Typing error
end Incorrect;
```

A goal of LibAdalang was to be usable in contexts such as syntactic editors, where the source can be incorrect, and to be able to fix such incorrect code or to automatically complete it. Of course, there are limitations to what can be achieved on incorrect code, since almost nothing can be assumed. LibAdalang provides operations to modify the underlying tree and the original source.

Moreover, LibAdalang provides a Python interface for developing rapid applications or interactively trying the interface.

4 Style and organization

ASIS

ASIS has a root package (Asis) that defines the basic entities used by the rest of the API. Child units group queries according to the structure (chapter and clauses) of the Ada reference manual: Asis.Declarations, Asis.Definitions, Asis.Expressions, Asis.Statements, etc.

Other packages are provided for initialization and loading of compilation units, accessing the source text of any element, etc.

The tree managed by ASIS is strictly the abstract syntax tree (AST) as defined by the syntax of the language; concrete elements that are not syntactic elements (comments, semi-colons, spaces) do not appear in the tree. It is possible to access the source corresponding to a node in the tree, but only as text. This makes it more complicated to develop applications like pretty printers that deal mainly with the physical appearance of the program [4].

LibAdalang

LibAdalang provides only two packages related to analysis: Libadalang.Common and Libadalang.Analysis. The root package Libadalang is almost empty and serves only as the parent of the hierarchy. Libadalang.Common groups general declarations of types and subprograms used in the rest of the API, including declarations intended only for the implementation of the library, and not for the user of LibAdalang. Libadalang.Analysis gathers all syntactic and semantic queries in a single package; as of this paper, the specification of Libadalang.Analysis contains 15992 raw lines, including 14154 lines in the visible part.

Other packages are provided for initialization and loading of compilation units, changing program text on the fly, etc.

In addition to syntactic elements, LibAdalang keeps all syntactic tokens, including *trivias* representing the concrete representation, like spacing in the source, semi-colons, comments, etc. The goal is to be able to manipulate the concrete representation of the program as well as its abstract structure.

5 Typing system

ASIS

All Ada elements are of a single type: Element. Subtypes are provided for documentation purposes, like:

```
subtype Declaration is element;
```

but since there are no constraints, there is no compile-time check that an element belongs to the expected subtype. The hierarchy of syntactic elements is accessible through a number of classification functions returning enumeration types that tell the precise "kind" of the element. For example, the function Element_Kind returns a value like A_Declaration, An_Expression, A_Statement… If the element is a declaration, the function Declaration_Kind returns A_Subtype_Declaration, A_Variable_Declaration, A_Constant_Declaration ...

Every query expects its argument to be of certain kinds, and checks it at run-time (and raises the exception Inappropriate_Element if the checks fails). This means that ASIS is strongly, but dynamically typed.

This comes as a surprise to many Ada users who are more accustomed to static strong typing. However, this simplifies navigating through the syntactic tree, since it is often not possible to foresee the exact nature of the result of a query. For example, it is very common in tools to navigate "up" the tree. A simple loop to find which procedure body encloses a certain element can be done simply, thanks to dynamic typing:

```
declare
  E : Asis.Element := Some_Query (...);
begin
  while declaration_kind (E)
    not in A_Procedure_Body_Declaration
  loop
     E := Enclosing_Element (E);
  end loop;
end;
```

LibAdalang

LibAdalang represents elements as a hierarchy of tagged types, rooted at Ada_Node:

```
type Ada_Node is tagged private;
type Expr is new Ada_Node with private;
type Name is new Expr with private;
…
```

The whole hierarchy of elements contains 373 different types. Queries require parameters of the appropriate type, but return values of the *specific* type Ada_Node. These values must be converted to the appropriate type using ad-hoc conversion functions, which raise an exception if their argument is not of the appropriate kind. Here is a typical example of this programming pattern:

```
case Kind (Node) is
  when Ada_Call_Expr =>
    for Assoc of As_Assoc_List
                 (F_Suffix (As_Call_Expr (Node)))
    loop
      ...
    end loop;
  ...
end case;
```

Here, Node is obtained from a previous query, and is of type Ada_Node, and its real kind is obtained by the Kind function. Since F_Suffix expects a parameter of type Call_Expr'Class, it must be converted by the special function As_Call_Expr. Of course, this function will raise an exception (Constraint_Error) if the parameter does not correspond to the expected type.

Despite the apparent stronger typed hierarchy of elements, LibAdalang is also dynamically typed: if a node does not belong to the expected kind for a query, it will be checked at run-time by the corresponding "As_..." function instead of the function itself. On the other hand, the stronger typing makes exploring the tree more difficult; for example, the simple loop of the previous example has to be replaced by the following recursive function:

```
function Enclosing_Proc (N : Ada_Node'Class)
  return Ada_Node'Class
is
begin
   if Kind (N) in
     Ada_Subp_Kind_Procedure | Ada_Subp_Body
  then
     return N;
   else
     return Enclosing_Proc (Parent (N));
   end if;
end Enclosing_Proc;
```

6 Tree traversal

Tree traversal is the process by which a program explores the whole program under analysis.

ASIS

ASIS provides a generic traversal function, which must be instantiated to provide the actual traversal function:

```
generic
  type State_Information is limited private;
  with procedure Pre_Operation
                    (Element :       Asis.Element;
                     Control : in out Traverse_Control;
                     State   : in out State_Information)
    is <>;

  with procedure Post_Operation
                   (Element :       Asis.Element;
                    Control : in out Traverse_Control;
                    State   : in out State_Information)
    is <>;
procedure Traverse_Element
               (Element :          Asis.Element;
                Control   : in out Traverse_Control;
                State     : in out State_Information);
```

The Pre_Operation procedure is called when entering a node in the AST, before traversing the children, while the Post_Operation procedure is called when returning to the node after traversing the children. In addition, a variable of the (user provided) type State_Information is passed along. This makes it convenient to initialize information when reaching a node, gathering information while traversing the children, and using the result when returning to the node.

Each of the operations returns a value of the enumeration type State_Information; possible values are Continue (normal case), Abandon_Children (child nodes are not traversed), Abandon_Siblings (return immediately to the parent node), and Terminate_Immediately.

LibAdalang

LibAdalang provides a traversal function where the processing of the node is provided as a pointer to an appropriate function:

```
function Traverse
 (Node  : Ada_Node'Class;
  Visit : access function (Node : Ada_Node'Class)
                          return Visit_Status)
return Visit_Status;
```

The Visit function is called when entering a node in the AST, before traversing the children. The enumeration type Visit_Status can take the values Into (continue normally), Over (child nodes are not traversed) and Stop. There is no equivalent to Abandon_Siblings.

There is no provided function to perform processing when returning to the node after traversing the children; if this is desired, the Visit function must manually invoke Traverse on child nodes and add the necessary processing after it returns. Since there is no associated data, information must be gathered in a global variable, or equivalent data structure.

7 Documentation

ASIS

The official documentation is the ASIS standard itself. It provides a good description of how to build an ASIS application and examples, in addition to the API itself. But

being an ISO standard, it is a copyrighted document that must be bought from ISO at usual ISO price (currently, CHF 198).

However, AdaCore's implementation comes with an ASIS user guide that provides appropriate guidance on how to build an ASIS application.

The API itself is self-documented. The naming convention of the structural queries follows strictly the names and the syntax used in the ARM, making it easy to find the desired function. For example, the syntax of an assignment in the ARM is given as:

*variable*_name := expression;

The corresponding structural queries will be:

```
function Assignment_Variable_Name
   (Statement : Asis.Statement)
    return     Asis.Expression;

function Assignment_Expression
   (Statement : Asis.Statement)
    return     Asis.Expression;
```

Each query states precisely the kinds of expected elements, and the kinds of the provided result. For example, the comments on the above Assignment_Variable_Name function state:

```
-- Statement - Specifies the assignment statement to query
-- Returns the expression that names the left hand side of the
-- assignment.
-- Appropriate Element_Kinds:
--     A_Statement
-- Appropriate Statement_Kinds:
--     An_Assignment_Statement
-- Returns Element_Kinds:
--     An_Expression
```

Structural queries have names starting with "Corresponding_", making it easy to read and understand. For example, the query used to find the declaration of a given name is called Corresponding_Name_Declaration.

LibAdalang

LibAdalang comes with a user guide, providing a detailed explanation of how to create an application, both in Python and Ada.

The API has a layout that shows that it is still work in progress: no header comments, lots of useless blank lines, poor indentation… The naming convention of queries bears no relationship to the reference manual; for example, the syntactic element Selector in Ada is called Suffix in LibAdalang. Many names use abbreviations whose meaning is far from obvious, when not misleading. For example, the query that returns the list of names that follow a with clause is named F_Packages… although a with clause can refer to units that are not packages!

Queries are divided into those that return "fields" of the underlying structure (i.e. structural queries) and those that return "properties" (i.e. semantic queries). The first ones have names that start in F_ and the second ones have names that start in P_, a convention that may be useful to the implementation, but makes the reading quite unnatural. The documentation (in the accompanying document or in the comments in the package) is often missing, or simply states what kind of nodes is contained in the corresponding field - often with misleading information. For example, the documentation of the F_Dest query for an assignment (the left-hand side of the assignment) mentions as possible fields Attribute_Ref, Char_Literal, and String_Literal, while actually these cannot appear as the destination of an assignment statement!

8 Extra functionalities

ASIS

The ASIS standard defines just an API. Helper utilities can be provided by the implementation, but there is no requirement to do so.

The AdaCore implementation provides a utility called Asistant that allows exercising interactively all queries of the API. It is very useful to understand the exact behaviour of the queries, but it is not intended as a way of quickly analysing a program.

As part of the distribution of AdaControl [5], Adalog provides a small utility called ptree that prints a semi-graphical representation of the syntactic tree, as obtained from ASIS; this is handy in understanding the structure of the AST.

LibAdalang

LibAdalang provides a Python API in addition to the Ada API. This can be used to exercise queries as well as for developing quickly and interactively small analysis tasks. Of course, the Ada programmer will prefer the Ada interface for applications with a long lifetime…

For quick development of a command line application, LibAdalang provides the generic package App, which can be instantiated with a procedure traversing the tree, and sets up automatically all the environment, including command line parameters analysis, setting of the environment, etc.

An associated project is LKQL (*LangKit Query Language*), a language intended to provide language queries on top of LibAdalang.

9 Pros and Cons

ASIS

Pros: ASIS works on legal code, after analysis by an Ada compiler that passes the validation. This provides great confidence that the program is analysed in conformance with the standard, and that the content of packages Standard and System match the ones used by the compiler. It comes with a complete documentation in the API, strongly linked to the Ada Reference Manual, making it easy to retrieve the necessary queries and to understand its effects.

Cons: The fact that ASIS works only on legal code and cannot change the tree or the corresponding source makes it

inappropriate for interactive applications, such as IDEs, where the source is incomplete or evolving. Although AdaCores's implementation processes all versions of Ada up to Ada 2012, an update of the standard to the latest version of Ada would be welcome.

LibAdalang

Pros: LibAdalang is able to work on incomplete/incorrect code and provides sophisticated support to the concrete representation of the program, as well as editing and modifying the original text. It processes the latest version of the language.

Cons: As there is no connection to the compiler, there is no guarantee that LibAdalang's view of the program corresponds the compiler's view or to the Ada standard. An analysis tool cannot rely on the fact that there is no diagnosis to trust that Ada rules are being obeyed; therefore the tool should be run only on programs that have been successfully compiled with a full Ada compiler.

The typing system and the distance between Ada's formal definition and the analysis packages, as well as the lack of a number of useful features, makes it less fit for deep analysis of Ada code.

The development of LibAdalang is fully under control of AdaCore without external review, and of course it is not a standard, nor expected to become one.

Conclusion

ASIS and LibAdalang cover different parts of the spectrum of code analysis tools. ASIS, thanks to its precise specification, its close connection to the Ada definition, and its guarantee of semantic correctness of the analysed code, is more appropriate for long lived tools, and especially tools expected to be used in demanding domains like instrumentation or control of safety critical software, The dynamic aspect of LibAdalang is well suited for all the tasks that require close connection to the source, user interaction, or dynamic modifications, like syntactic editors, code generators, and code transformation tools.

Given the fundamental differences in philosophy and typing systems between ASIS and LibAdalang, moving a tool from ASIS to LibAdalang cannot be done easily and would require a complete rewrite.

References

[1] ISO/IEC 15291: Information technology — Programming languages — Ada Semantic Interface Specification (ASIS)

[2] https://github.com/AdaCore/libadalang

[3] Goos, Gerhard; Winterstein, Georg (1980). "Towards a compiler front-end for Ada". Proceedings of the ACM-SIGPLAN symposium on Ada programming language. Annual International Conference on Ada. ACM-SIGPLAN. pp. 36–46.

[4] S. Rybin & A. Strohmeier : " About the Difficulties of Building a Pretty-Printer for Ada", proceedings of the Reliable Software Technologies — Ada-Europe 2002 conference, June 2002.

[5] https://github.com/Adalog-fr/Adacontrol

[6] https://github.com/AdaCore/langkit-query-language/blob/master/user_manual/source/language_reference.rst