The new semantic model in ASIS for Ada 2005

J-P. Rosen
Adalog
19-21 rue du 8 mai 1945
94110 ARCUEIL
FRANCE
+33 (1) 41 24 31 40
rosen@adalog.fr

T. Taft
SofCheck, Inc.
11 Cypress Drive
Burlington, MA 01803-4907
USA
+1 (781) 750-8068
stt@sofcheck.com

ABSTRACT
Following the new Ada 2005 standard, the ASIS interface has to be upgraded. In addition to supporting the representation of new features, ASIS has been extended with a *semantic* subsystem, where queries support the logical *views* of entities rather than their syntactic counterparts. This paper describes the new concepts introduced by this new subsystem.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Parsing*.

General Terms
Static analysis, standard, programming languages.

Keywords
Ada, ASIS, Ada 2005, Static analysis.

1. ASIS

1.1 General presentation

ASIS (Ada Semantic Interface Specification)[1] is an ISO standard that defines an interface between an Ada environment (i.e. compiler and tools) and any tool requiring information from it. It has been designed to be independent of underlying Ada implementations, and is currently supported by most compilers.

The many features of Ada make it difficult for tools to analyze the language; simplistic parsers do not provide enough information for any serious analysis. For example, given the following construct:

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

"A" can be a function call, with B as a parameter; an array variable (or constant) with B as an index; a type name in a conversion with B as a parameter; a parameterless function call (that returns an array) with B as an index; or the implicit dereference of an access variable that designates any of the previous elements. Disambiguating the construct requires type analysis, visibility analysis, and overload resolution. Requiring every tool to include such machinery would be a waste of effort, and it would be difficult to make sure that the tool does it right. On the other hand, this machinery exists in the compiler, and has been extensively checked by the ACATS validation suite and by years of use.

ASIS provides an interface to the decorated tree produced by the compiler. This means that it defines *structural queries* related to the structure of the tree, and *semantic queries* related to links between elements that do not appear directly in the syntactic constructs.

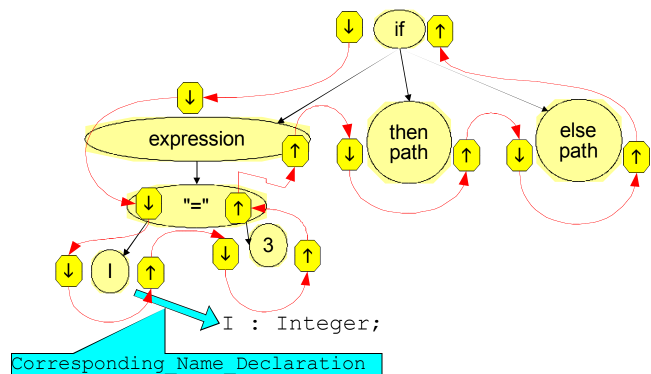
For example, if S is a statement element that contains a "**while** loop" statement, the query "While_Condition (S)" returns the expression that follows the **while** keyword. This is a structural query. If N is an expression element that contains a name, the query "Corresponding_Name_Declaration (N)" returns the declaration of N; since there is no *syntactic* connection between a name and its declaration, it is a *semantic* query. All semantic queries have a name that starts with "Corresponding_".

Using only syntactic queries, it is possible to traverse the whole tree of a program. However, it requires an extensive analysis of every Ada construct. To ease the job of the tools, ASIS provides a service to traverse the whole tree, in the form of a generic that traverses the tree in a top-down, left-to-right order. The formal procedure given as the *pre-procedure* is called on every node when the node is entered, and the one given as the *post-procedure* is called when leaving the node, after traversing all the child elements.

For example, given the following statement:

```
if I = 3 then
   …
else
   …
end if;
```

The corresponding tree is:



The down arrows show the places in the traversal where the pre-procedure is called on the corresponding node, while the up arrows show the places where the post-procedure is called. "Corresponding_Name_Declaration" is an example of a semantic query that will lead from the name "I", as found in the tree, to its declaration.

All nodes in the tree are of the same type, called "Element". Categorization queries provide detailed information about the element: for example, the "expression" node above will have an "Element_Kind" of "An_Expression". This can be refined by "Expression_Kind", which returns "A_Function_Call", etc.

Typically, an ASIS tool will simply instantiate this generic, and perform the desired analysis when encountering one of the appropriate nodes.

1.2 Syntactic improvements for Ada 2005

The principle of ASIS requires there be a structural query returning the sub-elements of every Ada construct. Since new constructs have been introduced in Ada 2005, it was necessary to introduce the corresponding structural queries.

There are a few additions in Ada 2005 that require the definition of new syntactic elements. These include the extended **return** statement, interface types, **raise** with message, null procedures, private and limited **with**, **null**-excluding subtypes, the prefixed notation (known as Object.Method notation), and the addition of [**not**] **overriding** to subprogram declarations. These required the addition of new queries whose specification (and use) are quite straightforward.

The generalization of anonymous access types required a change in how such a type is recognized. Previously, being an anonymous access was just a feature (a *trait* in ASIS-speak) of a formal parameter; but now since anonymous access types are allowed in many more places, they have been promoted to a full new kind of declaration. The previous mechanism has been kept as an obsolescent feature; its use in new ASIS applications is discouraged.

In many cases, such as new pragmas and attributes, supporting the new features required only adding some values to already existing enumeration types. Of course, the introduction of new standard packages did not require any change to ASIS.

In short, the new features should not represent a major perturbation for existing ASIS tools. Since most of the analysis is done by **case** statements driven by the kind of construct, a tool that chooses to use an implementation of ASIS for Ada 2005 while remaining compatible with ASIS for Ada 95 will simply need to add a "**when others** => **null**" branch (that covers nothing for Ada 95) in some places (and if it forgets to do so, the compiler will tell). If it chooses to handle the new constructs (and lose compatibility with ASIS for Ada 95), it will need to add the appropriate new branches to the **case** statements and use the appropriate queries, but no global restructuring of the tool will be necessary.

1.3 Why the syntactic interface is not enough

ASIS has proved itself to be useful for tool writers. However, tool writers who are focused more on the underlying semantics rather than surface syntax issues have found that the use of the syntax of the language as the fundamental organizing principle of the ASIS interface can complicate processing. When viewed at an appropriate, more semantic level of abstraction, such processing might be quite straightforward.

An example of the kind of information that is difficult to determine from looking at an abstract syntax tree, but is already determined by the compiler during its semantic analysis, is whether a given name or expression denotes a read-only view of an entity (a constant or a pure value), or a writable view of an entity (a variable). If one examines the language rules in subclause 3.3 of the Ada Standard for what sorts of constructs denote constant views of objects, one finds that it is a diverse collection, with a significant number of context sensitive caveats.

For example, consider the use of a name denoting a protected type which happens to be enclosed within the associated protected body, and also happens to be within a function immediately declared within the protected unit, as in the following example:

```
protected type PT is
   function F return Integer;
private
   Val : Integer;
end PT;

protected body PT is
   function F return Integer is
   begin
      return PT.Val;  -- What is this "PT"?
   end F;
end PT;
```

In this case, the name (*PT*) denotes a constant view of the *current instance* of the protected type. In other contexts the name might denote a writable view, or might denote the type itself. If the only question of interest to the tool writer is whether the given name is a reference to a read-only or writable view, then puzzling out all of the relevant context-sensitive rules is a significant waste of effort, particularly given that the compiler has already done all of the work itself.

For other examples of interesting properties of an entity that are not necessarily trivial to determine given only an abstract syntax tree for the construct that defines it, consider whether a given type is limited[1], or whether it is descended indirectly from Ada.Finalization.Controlled, or whether it contains a task. Consider whether a view of an object is aliased, or whether a given name refers to the limited or the full view of a package. The compiler has already determined this kind of information for its own purposes, and the ASIS Semantic Subsystem can therefore readily provide such information for use by the tool writer. This simplifies the tool writer's job, while also ensuring that the tool and the compiler agree about such properties.

The purpose of the semantic subsystem of the ASIS interface is to provide a uniform and consistent interface based primarily on the concepts of entities, views of entities, and declarations of views of entities, with the goal of capturing for the tool writer all of the work that the compiler has already performed in analyzing the program, including resolving overloaded names and constructs to their underlying meaning.

As an example of how the Semantic Subsytem can simplify processing, there is a simple query readily available on any given Object_View, Is_Constant_View, to tell whether the associated view is a constant or a variable. More generally, almost any interesting property of an entity, or view thereof, is directly available via a query in the Semantic Subsystem, independently of whether the property is explicitly specified with syntax on the view's declaration, or is inherited from some ancestor or

[1] An existing ASIS application, AdaControl, includes a function to determine if a type is limited using only the Syntactic Subsystem. It is 166 lines long.

progenitor entity, or is determined by the location where the reference occurs within the program.

2. ENTITIES AND VIEWS

The distinction between program *entities* (such as types, objects, packages, etc.) and *views* thereof, though it underlaid several of the concepts of Ada 83, was not clearly defined until Ada 95, and became even more important in Ada 2005 with the introduction of the limited view of a package. Each declaration in Ada declares some *view* of an *entity*, and usually the entity itself. A renaming declaration declares only a new view; the entity being renamed is generally declared by some prior declaration. In some cases there are multiple declarations for the same entity, with the first giving a partial or incomplete view, and the second giving the full or complete view. This is the case with incomplete and private types. Finally, in Ada 2005 there is the case of a **limited with** clause, which provides a *limited view* of a package, which includes an *incomplete view* of all the types declared in the visible part of (the full view of) the package.

Various properties of an entity can differ from one view to another. For example, in the partial view of a type, it might be an opaque private type, whereas in the full view, it might be an array type. Similarly, in one view of an object it might be a variable, while in another view, such as that provided by a formal **in** parameter of a by-reference type, it might be a constant. As another example, in one view of a function it might have two operands named Left and Right, while in another, defined by a subprogram renaming, the operands might be named L and R. In all of the above cases, there is only one program entity, but its properties vary depending on the view denoted by a given name.

The ASIS Semantic Subsystem deals with views of entities, and the answers provided by the various queries reflect the particular view of the entity. A query such as Is_Constant_View returns a value based on the particular name or expression from which the view was retrieved via Views.Expressions.Corresponding_View. Even though the underlying object might be a variable, if the view was retrieved from a name that gives only read-only access to the object, then Is_Constant_View would return True. Similarly, Is_Array of a view of a private type will return False even if the underlying type is an array type, if the view was retrieved from a name found in a part of the program with visibility only on the partial view.

3. A MORE STRONGLY TYPED MODEL

The Syntactic Subsystem of ASIS uses subtypes of the single type Element to represent all the various kinds of syntactic constructs that exist in Ada. Although various subtypes are provided (Statements, Declarations, etc.), they serve only for documentation purposes, since there is no constraint associated (Element is a private type). This makes it easy to declare a variable (of type Element) that walks down a structure, therefore being in turn, for example, a statement, then an expression that is part of the statement, then a function call that is part of the expression, etc. At run time, each query checks that the element passed to it is appropriate, and raises an exception if not. In a sense, the syntactic subsystem is strongly, but dynamically typed: no checking is possible at compile time, everything is checked at run-time.

A similar approach was considered in the Semantic Subsystem to represent the various kinds of views, entities, and declarations, but given that this was a completely new part of the standard, it seemed worth attempting a more strongly-typed approach to the interface. We considered both a set of related untagged derived types, as well as a hierarchy of tagged types, to represent the hierarchy of kinds of declarations and entities in the language, such as declarations, program units, and library units, and elementary types, scalar types, and numeric types. The untagged types had the advantage of the easy representation of a variable that could be used to hold a reference to any entity in a given type hierarchy, particularly compared to the relative difficulty of creating tagged *class-wide* variables, which must be initialized at the point of their declaration. However, the appeal of having the full power of the object-oriented features of the language, including interface types which were new to Ada 2005, overcame the concerns about declaring variables.

To address the issue associated with declaring variables, we defined a new kind of container called a *holder*, which holds at most one object of a given (sub)type, including an *indefinite* subtype, such as a class-wide subtype. Normally a variable of an indefinite subtype must be either constrained or initialized at its point of declaration. However, by declaring a *holder* for the given classsubtype, one can initialize the holder after its point of declaration with the result of calling a function that returns a result of any type *covered* by the class-wide type associated with the holder.

Once we had agreed upon the use of holders, we allowed ourselves to use the full power of the object-oriented features of the language to provide a natural, hierarchical, strongly-typed representation of the kinds of declarations and entities that appear in Ada. At the top of the hierarchy we have the View type. The class-wide type View'Class represents a view of any kind of program entity, including an object or value, a subtype, a package, a generic, an exception, or a callable entity such as a subprogram. Essentially anything that can be declared or denoted with a name or expression can be represented by an object of typeView'Class.

Using View'Class directly gives the tool writer maximum flexibility, but a relatively limited degree of strong compile-time type checking. We therefore provide additional types and their associated class-wide types which correspond to more specific kinds of entities, with the first level being Subtype_View'Class, Object_View'Class, Package_View'Class, and so on. In the case of subtypes, additional more restrictive types are defined to represent more narrowly-defined subclasses of subtypes. Hence, the tool writer may use the Access_To_Object'Class when they know they will be manipulating only views of such subtypes, or they may use Elementary_Subtype'Class when they are writing more general code to handle any sort of elementary subtype. There are also queries to check on the particular kind of subtype currently being manipulated, to decide what if any additional processing is appropriate.

For each kind of subtype view, there are operations appropriate to that kind defined in the ASIS Semantic Subsystem. Only operations that are meaningful on a given kind of subtype are defined for the corresponding descendant of Subtype_View, ensuring that many common coding errors will be caught when the tool is compiled, rather than when it is being applied to some new program. The type names used within the tool also provide useful and reliable documentation of the kinds of entities being manipulated by a given part of the tool. The Syntactic Subsystem can unfortunately not make such a promise, since all the subtypes of Element are interassignable and can be passed to any operation

defined on any subtype of Element. Essentially all the well known benefits of strong type checking, coupled with the flexibility of an object-oriented type hierarchy, are available to the user of the ASIS Semantic Subsystem.

4. VIEWS IN ASIS 2005, AND HOW TO USE THEM

As explained above, views are the fundamental and most important notion of the new semantic interface. The previous logical hierarchy of elements, which could only be enforced at run-time, is replaced by a number of interfaces (in the Ada 2005 sense). The inheritance hierarchy of these interfaces is as follows:

```
View
    Subtype_View
        Elementary_Subtype
            Scalar_Subtype
                Discrete_Subtype
            Access_Subtype
                Access_To_Object_Subtype
                Access_To_Subprogram_Subtype
        Composite_Subtype
            Array_Subtype
            Tagged_Subtype
    Object_View
        Access_Object_View
    Callable_View
    Package_View
    Generic_View
    Exception_View
    Statement_View
```

Since views are interfaces, the actual (specific) type of queries that return some kind of view is never visible to the user; only the primitive operations are. This gives a lot of freedom to the implementation for the actual type, since the only requirement is to implement the given interface.

Everything in an ASIS application starts from the syntactic queries. It is therefore necessary to have functions to go from a name to the corresponding view, and conversely. A view of an entity is obtained from a syntactic element by using the "Corresponding_View" query; note that the view "remembers" where it was obtained from, and that the original (syntactic) element can be retrieved by using the "Element_Denoting_View" query.

Each kind of view provides a number of queries that are appropriate to the view. Providing the complete list would be out of the scope of this paper, but the general principle is that they allow access to information that is hard to get from the syntactic constructs, such as the primitive operations of a type, whether two subtypes statically match, the static accessibility level of an object, etc.

A static evaluator is included, which provides the computed value of a static expression: given a construct like "A+B" where A is a named number of value 1, and B a named number of value 2, the Static_Discrete_Value query returns the value of the expression (i.e. 3). For enumerated types, the position number of the expression is returned. As surprising as it may seem, all serious ASIS applications had to rewrite their own evaluator to access the value of static expressions[2]!

Another important improvement is the ability to deal with constructs that could not be related to a declaration. For example, consider a tool that wants to make sure that calls do not use the default values of some parameters. With syntactic queries, it has to go from the call to the declaration of the called subprogram in order to get the list of parameters. Unfortunately, if the call is through an access to subprogram (or if it is a dispatching call), the called entity is not statically determinable; short of a declaration, it was not possible to retrieve the profile of the called entity – although this information was clearly available to the compiler.

5. OTHER ENTITIES OF THE SEMANTIC MODEL

Views correspond to entities that have a name and a declaration (except for anonymous ones). However, other kinds of Ada constructs also benefit from higher level queries than those provided by the previous version of ASIS. For example, given an occurrence of a name, determining whether it is declared in the visible or private part of a package would require traversing the whole declarative region that encloses the declaration to retrieve the exact place of the declaration. Once again, this information is obviously known to the compiler.

A "View_Declaration" is an abstraction of the declaration of a view as a whole. Unlike the corresponding syntactic construct, a "View_Declaration" corresponds to a single view, even if the corresponding syntactic element defines multiple views. e.g. given:

```
A, B : Integer;
```

this construct corresponds to only one Element in the syntactic world, which reflects how the program was written, but corresponds to two declarations (one for A and one for B) in the semantic subsystem that reflects the logical properties. Queries are provided for properties of view declarations that are difficult to get from the syntactic model, such as whether a declaration is overloadable, whether it is overridden (or overriding), etc. As for views, there is a hierarchy of view declarations:

```
View_Declaration
    Program_Unit
        Library_Item
```

In addition, the new model provides access to declarative regions. A "Declarative_Region" is an abstraction of a declarative region, and queries allow retrieving various information related to the region, such as the enclosing region of a declaration, whether two declarations are part of the same region, and which one occurs first. Moreover, a "Declarative_Region" is defined to include several "Region_Part"s (such as, for example, the visible and private parts of a package specification). Queries allow retrieving the various parts, conversely retrieving which declarative region a part belongs to, the list of declarations that are included in a region, etc.

---

[2] To be honest, the GNAT implementation of ASIS did provide a static evaluator, but it was a non-standard addition.

In Ada 95, several kinds of constructs (pragmas, representation clauses) were involved in the physical representation of types. Since ASIS95 followed strictly the syntactic structure, it had no single query that would return all the declarative items related to the representation of a type, something that is needed by many tools. Part of the Ada 2005 clean-up was to define *aspects*, a notion that gathers representation clauses, representation pragmas, etc. As a consequence, the semantic subsystem now defines a number of queries that permit retrieving all aspects related to a given view. A notable improvement is that this includes aspects inherited by derived types, something that was difficult to retrieve previously.

6. CONTAINERS

As explained above, the use of different types (as opposed to different *subtypes* as in the previous model) triggered the need for a kind of indefinite container that would hold only a single value, but unlike a regular indefinite variable, would not need to be initialized when created, and could hold values of different specific types during its lifetime. Since it was obvious that this need was not restricted to ASIS, this new container has been proposed for the next amendment of the language [2] as package Ada.Containers.Indefinite_Holders. It works like an Indefinite_Vector of one element, but since its size is always one, the specification is much simpler (iterators are not needed!). The implementation of this package does not require anything special in the language and it is expected that implementations will be provided without waiting for the amendment to be completed.

In addition to holders, a vector (i.e. an instantiation of Ada.Containers.Indefinite_Vectors) is provided for each kind of view. It would seem sufficient to provide these instantiations only for View'Class (since such containers could contain any type in the hierarchy), but doing so would require explicit conversions to call any primitive operations of a specialized view, thus reverting to run-time consistency checks and losing the benefits of the stronger typing model. By providing specialized containers that follow the hierarchy of views, the type checking happens when the value is put into the container, and all subsequent use will benefit from compile-time checking.

7. CONCLUSION

In its principles, ASIS has proven to be a very powerful and efficient companion to the compiler. Not only does it allow the development of analysis tools by developers not associated with any specific compiler vendor, but even compiler vendors have used it for some of their tools.

However, many ASIS developers were frustrated bcause they were unable to analyze some constructs due to their dynamic nature, or had to painfully reconstruct information from the syntactic queries when the compiler obviously had done this already.

The semantic subsystem plugs that hole, and will certainly allow improvements to existing tools as well as easier development of new tools. In addition, it provides a new paradigm for dealing with objects that represent various kinds of views which allow more compile-time checking than was possible for elements in the syntactic model. As a side effect, it also demonstrates the value of the new features provided by Ada 2005.

8. REFERENCES

[1] ISO/IEC 15291:1999. Information technology — Programming languages — Ada Semantic Interface Specification (ASIS)

[2] AI05-0069, available at http://www.ada-auth.org/AI05-SUMMARY.HTML