An OPTRAN-generated Front-End for Ada

Paul Keller, Thomas Maas

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 04/90

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication and will probably be copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the publisher, its distribution prior to publication should be limited to peer communication and specific requests.

An OPTRAN-generated Front-End for Ada

Paul Keller, Thomas Maas

FB 14 - Informatik
Universitaet des Saarlandes
D-6600 Saarbruecken
Federal Republic of Germany
E-Mail: pk@cs.uni-sb.de, maas@cs.uni-sb.de

ABSTRACT

Ada is a high-level imperative programming language with complex static semantics. In this paper we present a compiler front-end that translates Ada programs into DIANA trees - a standard intermediate form for Ada. In order to avoid the difficulties that arise from the ambiguous Ada syntax we designed a transformation system that maps the initial program tree into a more specific one. Semantic checks like overload resolution, type checking etc. are then performed on that tree.

The OPTRAN system developed at the Universitaet des Saarlandes was used to specify and implement the Ada front-end. Both the analysis of static semantics and the transformation of the initial program tree are described using the OPTRAN language which combines attributed tree grammars with sets of so-called transformation rules. The Ada project was a test case for the OPTRAN generator.

In this paper we give a detailed view on the Ada front-end and summarize the experience gained in specifying and implementing it.

1. OPTRAN System Overview

The OPTRAN-System is designed as a specification tool for attributed abstract syntax trees and transformations thereof. The structure of abstract syntax trees is described by a regular tree grammar. A set of attributes is associated with each node of a tree. These attributes may be used to collect context information in the tree.

For every production of the tree grammar there is a set of associated semantic rules specifying the functional dependencies between attributes.

Transformation rules are used to describe possible changes in the structure of the tree. Each transformation rule consists of two parts: The so-called left-hand side specifies syntactical and semantical conditions for the applications of the transformation rules, the right hand side consists of the specification of the output tree structure possibly augmented by so-called explicit semantic rules.
The syntactical condition of the left-hand side is described by a tree pattern. Additional contextual conditions may be specified as a boolean predicate over the attributes of the input tree pattern.

A transformation rule is said to be applicable at a node of a tree

- if there is a match for the input pattern at that node
- and the predicate is satisfied.

If a transformation rule is actually applied the matching part of the input pattern is replaced by the output pattern. In general this transformation changes the functional dependencies of the attributes in the tree. Therefore the values of the attributes for which the dependencies have actually changed (the so-called inconsistent attributes) have to be recomputed.

The OPTRAN specification of a tree transforming system (t-system) is called transformation unit (t-unit).

A t-unit consists of 5 parts:

- a tree grammar describing the structure of abstract syntax trees
- the association between the nodes of the tree and sets of attributes
- semantic rules associated with productions denoting the functional dependencies between attributes
- a set of transformation rules
- a user supplied strategy for the application of transformation rules.

OPTRAN is a batch-oriented system to precompute a t-system for a given t-unit. At runtime the generated t-system consumes an abstract syntax tree and performs the following actions:

- Initially the attribute evaluator computes the value of all attributes according to the functional dependencies in the t-unit.
- The tree analyzer searches the attributed syntax tree according to the specified strategy for a node where a tree template matches and the corresponding predicate is satisfied.
- The tree transformer applies the selected transformation rule to the tree.
- The attribute reevaluator recomputes the value of inconsistent attributes of the transformed tree.

It is possible to perform these tasks efficiently because of the extensive analysis of the static properties of a t-unit at generation time:

- The attribute (re)evaluator generator analyzes the attribute dependencies
- The tree analyzer generator analyzes the set of input patterns of the transformation rules and produces an efficient tree pattern matcher [We83]
- The tree transformer generator generates efficient tree transforming programs for the transformation rules [Wi81].

2. The Ada Front-End

In this section we give a survey of a front-end for Ada that was specified and implemented using the OPTRAN system.

The front-end translates Ada programs into their respective DIANA representations. The input of the front-end may consist of several Ada compilation units, i.e. packages, tasks, generic units, procedures or functions. The front-end produces DIANA trees, a standard form to represent the structural and static semantic properties of Ada programs in compilers (cf. [GWE83]). These DIANA trees might be processed by further components of an Ada programming environment, e.g. optimizers, compiler back-ends, syntax-driven editors etc. .

The Ada front-end consists of three components:

A scanner/parser unit checks the lexical and syntactical correctness of the Ada input. An operator tree t1 is built up to represent the given program.

Analysis of static semantics is done by two OPTRAN transformation systems: t-system I analyses the explicit and implicit declarations in the input program. The operator tree t1 is then transformed into a tree t2. All subtrees of t1 that represent an Ada name construct are substituted by trees whose operators specify which category the name belongs to. For example a subtree that describes an Ada name like f(x) is replaced by a tree that designates a function call, a type conversion, an array access etc. . This transformation process which we call **name class analysis** is driven by semantic information accumulated in attributes. The transformations manipulate large areas of the input tree since an Ada programmer must always use a name construct to designate any object of the language. Name class analysis considerably facilitates the specification of overload resolution in t-unit II.

The task of the second t-system is to check all applications of identifiers in names and expressions thereby performing type checking and testing some other semantic rules of Ada. The central issue is to resolve overloading. Moreover t-system II is the interface of the front-end to further components of a programming environment. It produces the DIANA external form of the given Ada program.

Every component of the front-end was constructed by generators using abstract descriptions. The functionality of the scanner/parser unit is determined by a string-to-tree grammar processed by the POPSY/POCO system (cf. [GPSW86]). The t-units for static semantic analysis include attributed tree grammars and, in the case of t-unit I, a set of transformation rules. From these specifications the OPTRAN system creates the two transformation systems. In fact the second system is merely an attribute evaluator.

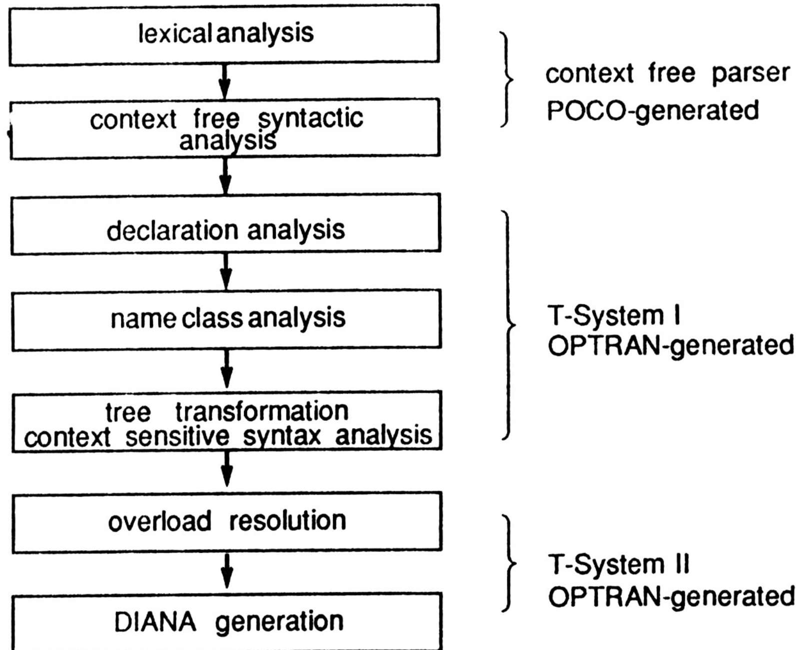

Fig. 1

Figure 1 shows the components of the front-end and the systems involved in their generation.

The following sections give a more detailed description of the semantic analysis phases and the experience gained in specifying and implementing them.

3. T-System I: Declaration and Name Class Analysis, Tree Transformation

The tasks of the first t-system are to do full Ada declaration analysis, to resolve syntactic unsignificance and to transform the attributed operator tree into a tree representing the special semantics of applied declarations in its structure.

3.1. Declaration Analysis

The declaration analysis phase of the t-system has to collect the information offered in the declarations and to keep it in a symbol table for later lookup. For every declared entity the symbol table has to provide a unique description for all its static properties. Besides these tasks common for all compilers the symbol table has to reflect special features of Ada [cf. Ada83] like:

- **block structured scopes**
- **packages**
- **generic units**
- **overloaded identifiers**
- **incomplete** and **textually separated declarations**
- **renaming declarations**
- **use clauses**

– **separate compilation**

The specific rules about the visibility, overloading and use of identifiers are very complex, changing subtly in different contexts. For every type of declaration the symbol table mechanism has to perform special semantic checks.
Besides explicit declarations some elements of Ada induce **implicit declarations**:

- derived type declarations may cause implicit derivation of overloaded subprograms
- label, block and loop names are implicitly declared at the end of the enclosing block.

For example the explicit declaration

type int **is new** integer;

implicitly declares the following overloaded operators:

function "="(left, right: int) **return** boolean;
function "/="(left, right: int) **return** boolean;
function "<"(left, right: int) **return** boolean;
function "<="(left, right: int) **return** boolean;
function ">"(left, right: int) **return** boolean;
function ">="(left, right: int) **return** boolean;
function "+"(right: int) **return** int;
function "–"(right: int) **return** int;
function "abs"(right: int) **return** int;
function "+"(left, right: int) **return** int;
function "–"(left, right: int) **return** int;
function "*"(left, right: int) **return** int;
function "/"(left, right: int) **return** int;
function "rem"(left, right: int) **return** int;
function "mod"(left, right: int) **return** int;
function "**"(left, right: int) **return** int;

For a detailed discussion about the situations where identifiers can be *renamed*, *hidden*, made *potentially visible* and later made *directly visible* and other problems complicating the visibility rules in Ada, see [Ada83], [Bac84] and [Ke90].

Symbol Table Attributes

The symbol table attributes *isymtab* and *ssymtab* (**i**nherited and **s**ynthesized **sym**bol **tab**le) are used to represent the logical sequence of declarations. These attributes are attached to every operator in the tree.
The inherited symbol table *isymtab* is initialized at the root of the operator tree to contain the predefined declarations of the Ada package **STANDARD** [cf. ADA83, Appendix C]
For those operators representing the various forms of declarations *isymtab* holds the symbol table state before the actual declaration, *ssymtab* represents the state after the declaration. The value of the *ssymtab* attribute is the symbol table before the declaration augmented with the new declaration if the semantic checks are passed successfully. Otherwise the symbol table contents are unchanged and an appropriate error message is emitted.*

---

* In fact most of the code of the semantic rules is checking for semantic errors. The error messages denote the error situation as specific as possible.

The 1086 semantic rules specifying declaration analysis are formulated to induce a **total evaluation order** for all symbol table attributes in the tree. This was neccessary as all the instances of symbol table attributes internally refer to a global symbol table structure. Because of this total order we can guarantee that the symbol table attributes are always in a consistent state. For a discussion about the special problems of global data structures see [Li86]. The data structures and algorithms used for declaration analysis together with the semantic checks are described in detail in [Ke90].

Symbol Table Lookup

In contrary to the bulk of specialized semantic routines used to enter a new declaration into the symbol table only 3 lookup functions are needed:

- the function *direct_visible* returns for a given designator at a position in the program text the set of all *directly visible* local and global declarations for that designator. All the rules of scope, of direct and of potential visibility, and of importing declarations from precompiled modules are taken into consideration.
- for a given designator the function *local_decls* returns the set of all local declarations with that designator within a given declarative region.
- the boolean function *is_nested* tests whether the actual position is nested within a given declarative region.

3.2. Name Class Analysis

Motivation

The majority of syntactic and semantic problems in the Ada language come from from the recursive nonterminal **name**. Throughout the language all kinds of references to a declared entity are specified using that nonterminal. The problem is that the alternatives derivable from **name** cannot be distinguished by a context free parser. The syntax of [Ada83] is augmented by semantic annotations.

When removing these annotations the syntax might look like:*

```
primary              ::=  ... | name | function_call | type_conversion
type_conversion      ::=  name ( expression )
type_mark            ::=  name
function_call        ::=  name [(expression {, expression })]
name                 ::=  prefix (expression {, expression}) | ...
indexed_component    ::=  name (expression {, expression })
slice                ::=  prefix (discrete_range)
procedure_call       ::=  name [(expression {, expression })] ;
expression           ::=  ... | primary
selected_component   ::=  prefix . selector
selector             ::=  simple_name | character_literal | operator_symbol | all
prefix               ::=  name | function_call
```

In order to be able to use a context free parser we had to remove the ambiguities in the grammar. In contrast to the grammar in [Ada83] the modified syntax does not distinguish between the above alternatives.

* Some alternatives are omitted. Square brackets enclose options, braces enclose zero ore more repetitions.

All forms of

name ::= name [(sequence of expressions)]

are initially represented by the operator *apply*.

Bottom-up name class pass

The attribute *snameclass* (**s**ynthesized **name class**) was introduced to compute the semantical class of name subtrees. These classes specify whether a name subtree is semantically a procedure call, a type conversion, etc. For every operator within a name subtree the *snameclass* attribute contains the class of the name and a set of references to those declarations being candidates for identification.

Three cases may shine up:

- Name class analysis can determine a unique name class and exactly one declaration for each component of a name subtree: there are no ambiguities within the name tree. Name class analysis has **identified** every component.
- Name class analysis can determine a unique name class for each component of a name subtree. For at least one component more than one declaration was found: the components of the name subtree can only be identified by overload resolution.
- The Ada input is semantically wrong. Appropriate error messages are emitted.

3.3. Tree transformation

The last phase of t-system I is the tree transformation phase. The purpose of this phase three-fold:

- Replacement of semantically ambiguous operators like *apply* by operators corresponding to the grammar.
- Due to the lack of context sensitivity in the [Ada83] syntax analysis phase some syntax properties are not checked by the parser. These **context sensitive*** syntactic checks are performed by the tree pattern matcher.
- Normalizations regarding the usage of Ada relational operators: Ada operators are defined to be equivalent to functions. Thus usage of Ada operators in conventional infix notation and in function call notation is semantically equivalent. Therefore the transformations normalize all applications of operators to use the function call notation.

The operator tree is searched for applications of application rules according to a top-down left-to-right strategy. In cases where more than one transformation rule is applicable the rule with the most specific input pattern is chosen. If two patterns are incomparable the textually preceding rule is chosen (cf. [Li88]).

* as defined by Chomsky

Examples

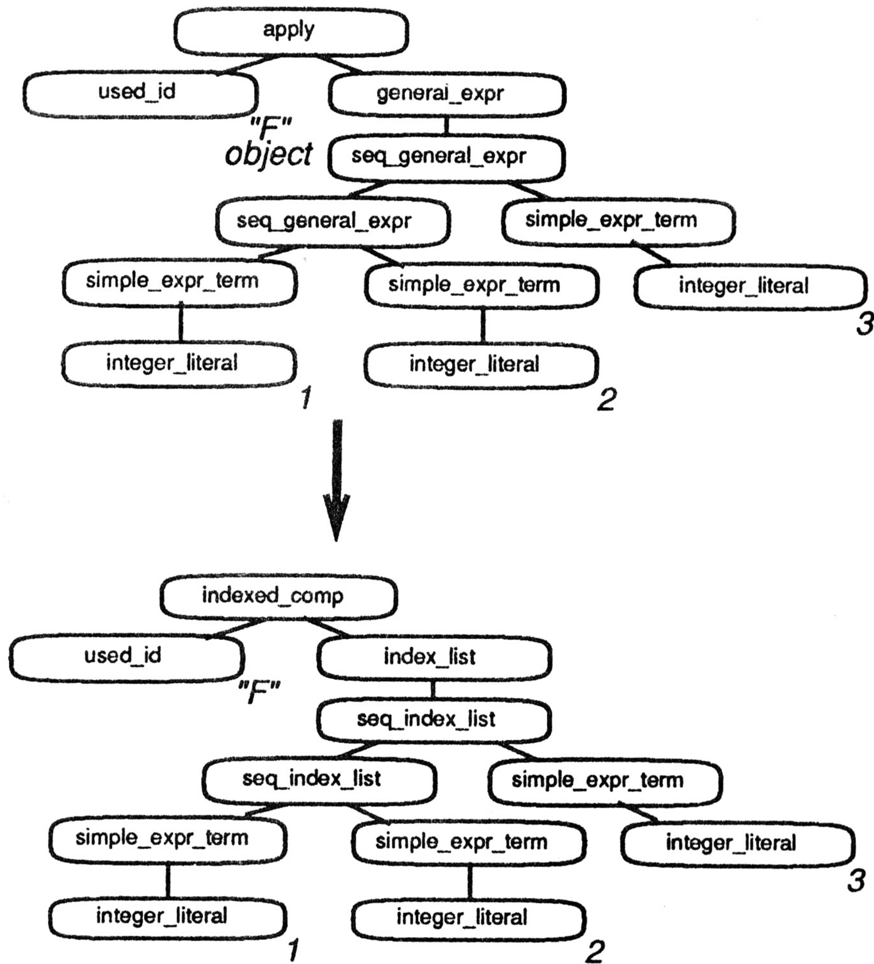

Fig. 2

Removing unspecific syntax:

Let *f(1,2,3)* be a component of the 3-dimensional array *f*. Fig. 2 shows the transformation of the corresponding apply subtree into a tree representing array indexing.

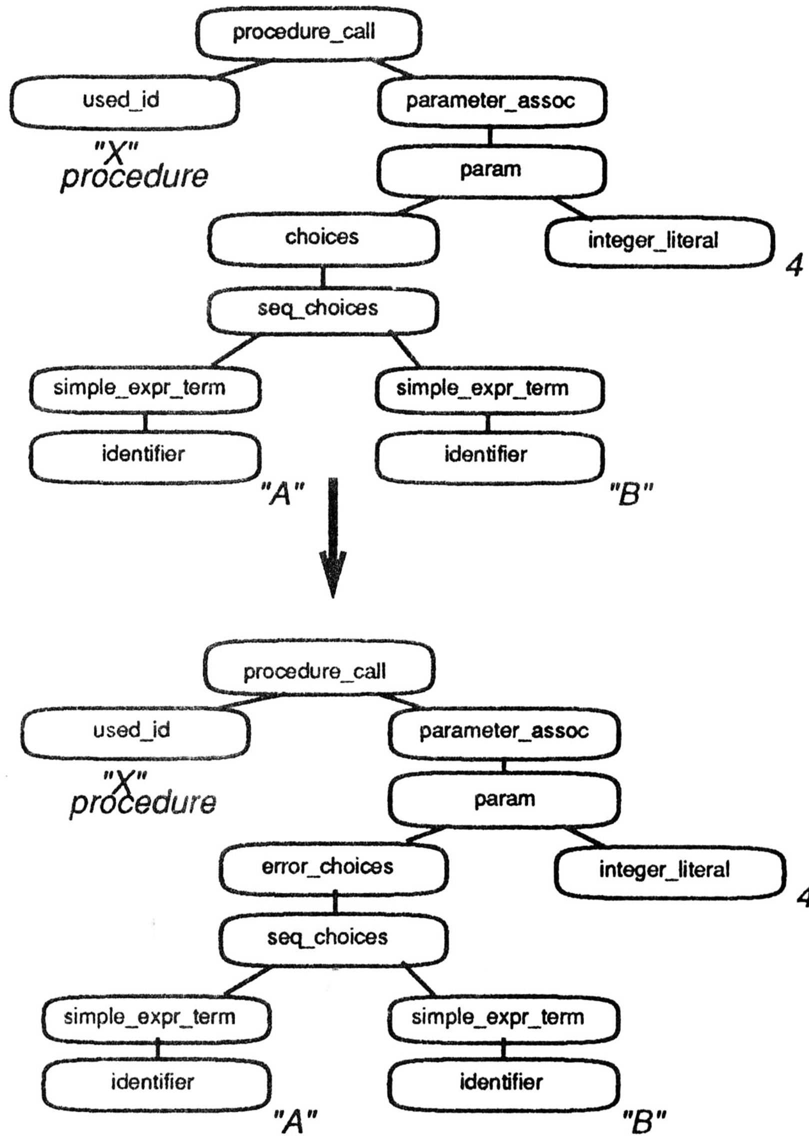

Fig. 3

Context sensitive syntax check:

The parser cannot distinguish between record and aggregate syntax and procedure calls. For a *variant record* **X** the program fragment **X(a|b => 4)** would define a default value of 4 for the *discriminant record components* **a** and **b**. But this notation would be illegal if **X** were a procedure. Fig. 3 shows how the transformation creates an specific operator indicating that error situation.

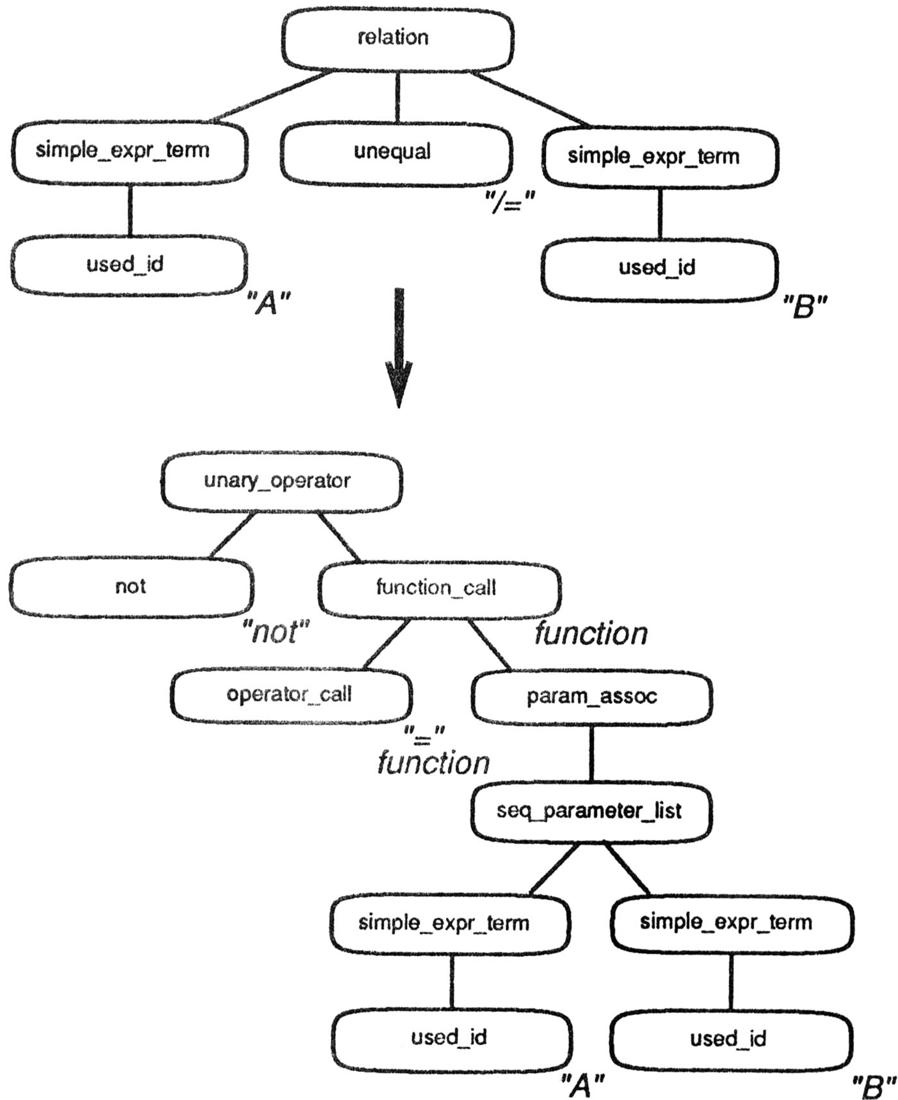

Fig. 4

Tree normalization:

Fig. 4 shows how the program fragment **a /= b** is transformed into the semantically equivalent form **not("="(a, b))**.

4. T-System II: Overload Resolution and Diana Generation

The tasks of the second t-system which is in fact an attribute evaluator are to identify the meaning of overloaded entities in the given Ada program and to build up its DIANA external form.

An identifier is overloaded if it can have several alternative meanings at a given point in the program text. Overloading in Ada can occur in one of the following forms (cf. [Ada83]):

- Identifiers of subprograms, i.e. **procedures** and **functions**, can be overloaded by those of other subprograms, by enumeration literals and by identifiers of entries.

The declaration of overloaded subprograms is allowed in the same scope if they differ in their parameter and result type profiles.

- Identifiers of **entries** and **enumeration literals** can analogously be overloaded by other entries or enumeration literals and by subprograms. The parameter and result type profile of an enumeration literal can be determined by interpreting the literal as a parameterless function whose result type is the type of the literal.
- The **literal null** of an access type - it designates an empty access value - overloads *null* literals of other acces types.
- **Aggregates** are used to explicitly state values of record or array types. An aggregate is a so-called basic operation that is created implicitly by declaring a record or array type. This operation is anonymous, i.e. it has no identifier. Aggregates of different types can be distinguished neither syntactically nor by an identifier. Therefore an aggregate is overloaded if it appears at a point in the program text where several record or array types are visible.
- An **allocator** *new* x creates a new object of type x and returns an access value for this object. It is an anonymous basic operation. An allocator *new* x is overloaded if it appears at a point in the program text where several access types are visible which all designate the type x.
- **String literals** are also overloaded basic operations.

There are two situations in which the Ada front-end must resolve overloading:

- Given is an application of an identifier x and a set D, $|D| > 1$, of declarations, which are visible at that point in the program text. D contains declarations of subprograms and/or entries and/or enumeration literals.
- Given is an application of an anonymous basic operation y (aggregate, allocator, string literal) or the literal *null* and there is a set T of visible types, $|T| > 1$, where every element of T might be the result type of y.

An Ada compiler must find the correct declaration or type in the set D or T by considering the context of the application. The compiler must state an error if none of the elements in D resp. T can be applied in this context. Both the outer and the inner context must be considered. In the example of an assignment A:=f(x) where A is a variable, f an overloaded function and x the actual parameter as well the type of A as the possible types of x must be used to determine the effective meaning of f. Moreover the inner and the outer context can themselves be overloaded.

Several algorithms for resolving overloading in preliminary Ada were published, e.g. [Bak82], [Cor81], [GaR80], [Per80]. The method used in our front-end is based on the algorithm of Persch et. al. [Per80]*. It analyses the operator tree of an Ada expression in a bottom-up pass followed by a top-down pass. Persch et. al. see all items in an expression tree as applications of functions. Initially an operator op is associated with the set of visible declarations for the function at op. The bottom-up pass deletes from the set those functions whose formal parameter types do not agree with the possible types of the actual parameters. The top-down pass then removes every function whose result type is not the one expected from the context. After completion of the second pass the set at every operator in the expression tree must contain exactly one element which is the actual meaning of the respective function call. Otherwise the expression is faulty.

---

* For a more detailed discussion of the different algorithms see [Ma88]

Persch et. al. use a model attribute grammar to describe their method for overload resolution. The grammar consists of one production that defines the structure of a function call and evaluation rules for two attributes that are used to collect the sets mentioned above. The attribute dependencies determine an evaluation in two passes. We transferred the method of Persch et. al. to the Ada grammar we used for the front-end (about 550 productions). The attribute grammar for Ada was specified in five phases:

Initialization

The attribute grammar for the second t-unit was derived from that of t-unit I by removing all *apply*-related operators. The attributes and semantic rules to build up the symbol table had to be installed in t-unit II once again because the OPTRAN system has no interface yet to take over values of attribute instances from one t-system to another.

The generation of the DIANA external form was already specified in the initial grammar. This considerably helped to check every developmental step of the front-end.

Bottom-up Pass

The attribute *spm* (**s**ynthesized **p**ossible **m**eanings) was introduced to collect the sets of possible declarations or types at an operator. The attribute occurences of *spm* and their semantic rules could not be specified in one step because the subgrammar for names, expressions and their contexts includes about 270 productions. Every occurence of *spm* must be computed according to the specific semantic conditions for every construct that may appear in an Ada expression (c.f. rules for function call, slice, aggregate, allocator, array access etc.). The individual constructs were stepwise integrated in the bottom-up pass beginning with the operators that may be the leaves of an expression tree. In several cases supporting attributes were introduced, e.g. to collect overloaded parameter and result type profiles. For names this differentiated specification process was possible because all apply operators are replaced by t-system I. After every step the front-end was generated once more.

The bottom-up pass had to be adjusted to some changes on preliminary Ada made in the language definition in 1983 [Ada83]. The type of aggregates and allocators must solely be determined from the context not considering the arguments. For example to resolve overloading of aggregates information on individual component associations must not be used. This arises the question what type an aggregate or allocator synthesizes in the bottom-up pass. We chose to define two types for this purpose, *universal_aggregate* and *universal_allocator*. They are different from every other type in a given program. In comparing types *universal_aggregate* matches every record or array type, *universal_allocator* every access type.

Numeric literals are not overloaded in Ada '83. Instead they are of the predefined types *universal_integer* resp. *universal_real*. Therefore implicit type conversions had to be integrated into overload resolution.

The development of the bottom-up pass was completed by designing an error reporting system that generates specific error messages (about 70 for overload resolution) and avoids follow-up messages.

Top-down Pass

The attribute *imeaning* (**i**nherited **meaning**) was introduced to collect the actual meaning of an operator. The top-down pass starts at productions where the right-hand side does not contain names or expressions but the production itself does not belong to the expression subgrammar, e.g. productions for assignment, abort statement, representation clauses etc. . The operators defined in these productions may be the roots of

subtrees on which overload resolution is applied. Again specific semantic rules had to be designed to compute a unique meaning at the root operator. For example for an assignment it is checked whether a unique type was synthesized on one of the sides of the statement. This type determines the meaning on the other side. So overloading can be resolved even on the left-hand side of an assignment.

OPTRAN allows for the use of local attributes in a t-unit. Local attributes are associated with a production rather than a nonterminal (cf. [Moe86b]). This feature proved very useful in the specification process for the Ada front-end. Temporary results of computations can be stored to be used by more than one semantic rule. T-unit II includes about 230 local attributes for overload resolution.

Evaluation of Static Expressions

As will be shown below the evaluation of static expressions for discrete types is a necessary part of overload resolution (cf. [Ada83], [Ma88]). The evaluation can be done following the top-down pass. The front-end must simulate the application of all predefined operators and Ada attributes on discrete types.

Analysis of Aggregates for Variable Record Types

The 2-pass scheme cannot be used to analyse aggregates of variable record types. Variable record types depend on one or more discriminants. An aggregate for such a type has a different structure depending on the discrimant values. It is a-priori unknown where the expressions that define values of discriminants are situated in the aggregate because named and *others* associations can be used beside positional ones. An Ada compiler must locate and evaluate the discriminant expressions to determine the structure of the aggregate which allows for resolving overloading in its components.

The 2-pass scheme extended by a bottom-up pass for static evaluation leads to an attribute grammar with cyclic dependencies. Consider the components of an aggregate: The start of the static evaluation pass depends on the presence of unique meanings at all operators of the expression tree. On the other hand not before the static evaluation of discriminant values is completed can the structure of the aggregate be determined which leads to unique meanings in the expression tree. For the Ada front-end we specified an algorithm that analyses an aggregate in one left-to-right traversal thereby avoiding cyclic dependencies.

It can be shown that any tree of the attribute grammar for overload resolution is still evaluable in two passes (cf. [Ma88]). The algorithm of Persch et. al. merely computes values of attribute instances in the up-visit of the first depth-first pass and in the down-visit of the second one. The computations that arise from static expression evaluation and from aggregate analysis can be performed during the up-visit of the second pass. However aggregate analysis requires an evaluation from left to right whereas the algorithm of Persch et. al. allows for evaluation in both directions. So the attribute grammar for overload resolution in Ada belongs to the class (L-AG; L-AG) which is included in the class of absolutely noncircular grammars, the class accepted by the OPTRAN system.

---

* For a detailed discussion see [Ma88]

5. Implementation data

on SUN 3/160, 16 MB memory, running UNIX 4.2 bsd, SunOS 3.4

Scanner/Parser:

String-to-tree grammar	2000 lines
300 lines Lexical description	300 lines
CPU-Time for Generation	20 minutes

Transformation-Unit I:

OPTRAN-Specification	14300 lines

including:
- 569 Productions
- 417 Operators
- 223 Nonterminals
- 20 Attributes
- 297 Local Attributes
- 4211 Applications of Semantic Rules
- 116 Transformation Templates

Semantic Rules (Pascal)	6200 lines
Generated Pascal-Source	26000 lines
Executable Transformer	730 kbytes
CPU-Time for Generation	45 minutes

Transformation-Unit II:

OPTRAN-Specification	15000 lines

including:
- 550 Productions
- 400 Operators
- 223 Nonterminals
- 35 Attributes
- 230 Local Attributes
- 4950 Applications of Semantic Rules
- 0 Transformation Templates

Semantic Rules (Pascal)	12000 lines
Generated Pascal-Source	26300 lines
Executable "Transformer"	780 kbytes
CPU-Time for Generation	45 minutes

6. Summary

We have shown how an Ada front-end was specified and implemented using the OPTRAN generator. The development of the front-end was a test for the OPTRAN system. We now summarize the experience gained in using the generator:

- The OPTRAN language has the power to express the complex static semantics of Ada. Both attributed grammars used in the t-units are included in the class of absolutely noncircular grammars which is accepted by the OPTRAN generator.
- The OPTRAN generator is capable of processing an input of the size required for Ada.
- The technique to use a generating system processing abstract descriptions proved an advantage for the development of the front-end. Analysis of static semantics could be shaped according to the abstract syntax of Ada. Subphases of the analysis could be introduced step-by-step. In every stage of the development new versions of the front-end were generated and testet. Moreover the consistency and completeness checks performed by OPTRAN at generator time were helpful to pin down errors in the t-units.
- Due to the large scale of the task at hand the benefit mentioned above was crucial for the feasibility of the project. But this advantage could only be exploited because the first t-system transforms the inital operator trees thereby establishing a syntax for Ada that is considerably more specific than the one delivered by the parser. The ambiguities of the Ada syntax as defined in [Ada83] would not have allowed for applying specific semantic rules to the individual constructs. Instead large general rules would have had to analyse constructs that are syntactically similar but semantically different.
- The OPTRAN feature of local attributes proved very helpful. Local attributes are widely used in both t-units.
- The compilation speed of the front-end is not optimal. The simple interface between t-systems that cannot hand over values of attribute instances is one reason thereof. The second t-system must build up the symbol table once again.
  Intensive profiling (cf. [Gra83]) shows that a high percentage of the running time is spent during initialization of internal data structures. This is due to the fact that this phase is implemented using Pascal file-io.
- We found it difficult to deal with the large attributed grammar for Ada without being supported by the generator. For example to introduce a widely used inherited attribute an extensive search had to be conducted for the productions where the attribute occurences must be computed by semantic rules. Generally it would be helpful to be supported in such a project by tools to deal with large grammars, e.g. an interactive system using a grammar graph (cf. [Moe86a]).

The Ada front-end and the OPTRAN system are operational and can be demonstrated.

References

[Ada83] Reference Manual for the Ada Programming Language, ANSI / MIL-STD 1815 A, 1983

[Bac84] Bach, I. **Unorthogonalities in the Identification Rules in Ada**, in *ACM Ada Letters*, Vol IV, March 1984

[Bak82] Baker, T.P. **A One-Pass Algorithm for Overload Resolution in Ada**, in *ACM Transactions on Programming Languages and Systems*, Vol. 4, No 4, October 1982, pp. 601-614

[Cor81] Cormack, G.V. **An Algorithm for the Selection of Overloaded Functions in Ada**, in *ACM Sigplan Notices 16*, February 1981, pp. 48-52

[GPSW86] Greim, M., Pistorius, St., Solsbacher, M. and Weisgerber, B., **POPSY and OPTRAN-Manual**, in *ESPRIT: PROSPECTRA-Project Report, S.1.6-R-3.0, Technical Report No. A 08/86*, FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1986.

[GaR80] Ganzinger, H., Ripken, K. **Operator Identification in Ada: Formal Specification, Complexity, and Concrete Implementation**, in *ACM Sigplan Notices 15*, February 1980, pp. 30-42

[GWE83] Goos, G. Wulf, W.A., Evans Jr., E., Butler, K.J. (Edit.) **DIANA An Intermediate Language for Ada**, Springer Verlag, Berlin 1983

[Gra82] Graham, S.L., Kessler, P.B., McKusick, M.K., **gprof: A Call Graph Execution Profiler**, in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, ACM Sigplan Notices, Vol. 17, No. 6, June 1982, pp. 120-126.

[Ke90] Keller, P., **Spezifikation und Implementierung eines Ada-Front-Ends mittels Uebersetzer-erzeugender Systeme, Teil I: Deklarations- und Nameclass-analyse**, Diploma Thesis (in german), FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1990.

[Li86] Lipps, P., **Komplexe Attribute - Mechanismen zur Verwaltung und Berechnung in einem baumtransformierenden System**, Diploma Thesis (in german), FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1986.

[Li88] Lipps, P., Moencke, U., Wilhelm, R., **OPTRAN - A Language/System for the Specification of Program Transformations: System Overview and Experiences**, in *Lecture Notes in Computer Science*, Springer, October 1988, pp. 10-14.

[Ma88] Maas, T., **Spezifikation und Implementierung eines Ada-Front-Ends mittels Uebersetzer-erzeugender Systeme, Teil II: Aufloesung der Ueberladung und Erzeugung der DIANA-Form**, Diploma Thesis (in german), FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1988.

[Moe86a] Moencke, U., **Grammar Flow Analysis**, in *ESPRIT: PROSPECTRA-Project Report S.1.3-R-2.1*, FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1986.

[Moe86b] Moencke, U., **Production Local Attributes**, in *ESPRIT: PROSPECTRA-Project Study Notes S.1.3-SN-3.0*, FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1986.

[Per80] Persch, G., Winterstein, G., Dausmann, M., Drossopoulou, S. **Overloading in Preliminary Ada**, in *ACM Sigplan Notices 15*, November 1980, pp. 47-56

[We83] Weisgerber B., **Attributierte Transformationsgrammtiken: Die Baumanalyse und Untersuchungen zu Transformationsstrategien**, Diploma Thesis (in

german), FB 14 - Informatik, Universitaet des Saarlandes, Saarbruecken, 1983.

[Wi81] Wilhelm, R., **A Modified Tree-To-Tree Correction Problem**, in *Information Processing Letters 12(3)*, 1981, pp. 127-132.