GNAT: The GNU Ada Compiler

June, 2004

Copyright (C) Javier Miranda and Edmond Schonberg
jmiranda@iuma.ulpgc.es, schonberg@cs.nyu.edu

Applied Microelectronics Research Institute
University of Las Palmas de Gran Canaria
Canary Islands
Spain

Computer Science Department
New York University
U.S.A.

Contents

List of Figures

Preface

The GNAT compilation system is a full implementation of Ada95 for a variety of machines and operating systems. GNAT is part of the GCC suite of compilers, and as such it is distributed under the GNU Public licence, with the suitable modified library licences that allow its full use in industrial contexts.

The GNAT system was first developed as an academic project at New York University (its was originally the acronym for the GNU NYU Ada Translator, but it is now a name with no encrypted meaning). Ada Core has taken over maintenance and development of the system since 1995. A version of the system is now part of the GCC sources, which has made Ada 95 into one of the core GCC languages.

GNAT is nowadays a mature technology used for large-scale industrial projects, as well as research and education in compiler technology. Though the system continues to evolve as it is ported to new targets, as more sophisticated optimizations are implemented (and as occasional bugs are fixed), the architecture of GNAT has been stable for a number of years. The purpose of the present document is to make this architecture more accessible to users and researchers. The sources of GNAT are very carefully documented, but a program of half-a-million lines is a daunting object to approach, and we hope that this document will make this approach easier.

Understanding a compilation system requires mastering two complementary subjects: the semantics of the programming language being implemented, and the algorithms that realize the translation and implement the run-time semantics. We have structured this document to serve as bridge between the two entities that are the ultimate authorities on each: the Ada95 reference manual, and the full sources of GNAT. We expect that the user of this document will navigate between it, the ARM, and GNAT, as need and curiosity dictate. For this reason the document has extremely numerous links to both, integrating them into a hypertext that is flexible, reasonably complete, and easy to navigate.

The code generator for GNAT is the GCC back-end, whose portability has allowed GNAT to be implemented on many targets. The architecture of GCC has been the subject of many publications, and we do not discuss any of it here. Therefore, this book is concerned only with the front-end of a full compilation system. This is large enough a task!

The standard description of a compiler front-end distinguishes between the context-free portions of the translation (lexical and syntactic translation), and the context-sensitive phase (static semantics). In this book we focus mostly on the second phase, for reasons of space, technical interest, and personal competence.

Audience

The book should be of interest to software practitioners interested in compiler technology, language experts who want to examine the implementation of complex constructs in a modern programming language, compiler writers looking for ideas, and software engineers interested in Ada95, in concurrency, in distributed programming and in real-time systems. Finally, the book will be useful to those who want to experiment with language extensions, and want to modify portions of GNAT to implement new constructs in Ada95 or the forthcoming Ada 2005 revision of the language.

In an educational context, the GNAT front-end is an interesting adjunct to courses in programming languages, compilers, and operating systems. The GNAT front-end translates a modern, complex imperative language with a rich type system, object-oriented features, and genericity. These central aspects of modern languages (Ada95, C++, Java, C#, etc.) are seldom discussed in detail in texts on Compilers. The user of this book will find therein a summary of the central implementation choices made in GNAT, and pointers to the detailed algorithms in the sources.

The GNAT run-time supports concurrency (tasking, protected objects) on a variety of operating systems, by means of a mostly-target-independent interface, whose primitives are close to those of the POSIX standard. As such, it can be a useful adjunct in a discussion of Operating Systems, of the semantics of concurrency, and on the efficiency of various synchronization primitives.

The book is not intended to be a stand-alone text in Compilers, nor a reference in which to learn Ada. The reader is assumed to have some familiarity with the language, and with basic compilation techniques, such as would be a found in a senior course in Programming Languages. We have included brief descriptions of the more interesting constructs in the language, to motivate the issues of implementation that we want to discuss. We have also included numerous language fragments to illustrate specific translation techniques. Many of these fragments present not just a source program, but its rewriting and expansion into a simpler form that is more amenable to translation into machine language.

Use of this Book

In order to facilitate the use of the book, it is distributed in several formats: HyperText Markup Language (HTML), PostScript (PS), and Portable Document Format (PDF). The HyperText version is the recommended format because it is linked with the GNAT sources, what allows the reader to analyze additional details not discussed in this book. The other formats facilitate its printing.

Structure of the Book

The book is structured in four parts:

- **Part I: Introduction**. The three chapters in this part present an overview of the GNAT compilation system, the architecture of the front-end, the data-structures used in the abstract representation of a program, and the error diagnostic and recovery techniques implemented in the GNAT parser and semantic analyzer.
- **Part II: Semantic Analysis**. Here we discuss the implementation strategies for the more interesting and complex aspects of the language, from the point of view of their translation into a low-level target independent representation. The topics include the analysis of scope and visibility, type checking and overloading, discriminants, generics, and freezing rules.
- **Part III: Expansion**. This part describes the first step in the synthesis of an object program, namely the transformation of the first target-independent representation into a simpler one, whose semantic level is close to that of C and which is therefore easier to translate into machine language. The complexity of this phase is a reflection of the richness of modern programming languages. Constructs that present interesting expansion challenges,

and which we discuss at length, include aggregates, tagged types, controlled types, protected objects, tasks, and everything having to do with intertask communication.

- **Part IV: The Run-Time**. In this part, we discuss the structure of the run-time system. We discuss the following in some detail: tasking (creation, activation, termination, abortion), rendezvous, protected objects, clocks and delay statements, exceptions, interrupts, and finally asynchronous transfer of control.

- **Part V: Appendix**. This appendix describes how to modify the GNAT front-end to implement language extensions. We sketch the techniques necessary to add keywords, pragmas, attributes, and new constructs to the language.

As explained above, the book does not discuss the implementation of the scanner and the parser, which are more conventional portions of any compiler. The interested reader will nevertheless find much elegant algorithms in those sections of the sources. The scanner is engineered to handle multiple character encodings, and the parser has what is considered to be the best set of error recovery strategies of any Ada compiler in use. We trust that compiler engineers will find a wealth of ideas in those sources.

Acknowledgements

The GNAT compiler is the product of several hundred person-years of work, starting with the NYU team that created the first validated Ada83 translator more than 20 years ago, and continuing today with the dedicated and enthusiastic members of Ada Core Technologies, and the myriad supportive users of GNAT whose suggestions keep improving the system. It is impractical to aknowledge all of the above by name, but we must express our very special thanks and admiration for Robert Dewar, chief architect, team leader, creator of some of the most interesting algorithms in GNAT, tireless enforcer of good programming practices, and an unsurpassable example of how to write impeccable software.

Part of this work has been developed under grant of the Spanish Government, reference code PR2002-0290.

Short Biography

Edmond Schonberg is professor and deputy Chair of the Department of Computer Science at New York University, and one of the founders of Ada Core Technologies. He has a PhD in Physics from The University of Chicago, and a BA in piano performance from the National Conservatory in Lima, Peru. His research interests are in programming languages, compilers, and optimization. He was part of the team that created Ada/Ed, the first complete translator for Ada83, and was one of the leaders of the GNAT project at New York University, that built the first prototype compiler for Ada95.

Javier Miranda studied Computer Science Engineering at the University of Las Palmas de Gran Canaria. He finished his studies on 1990 by implementing a Modula-2 compiler and went to the Technical University of Madrid to do his PhD under the direction of Angel Alvarez and Sergio Arévalo, on the design of an Ada extension for programming distributed systems. The experimental integration of this work into the GNAT compiler led him to become familiar with the internal details of the GNU Ada compiler. On 2003 he went to New York to collaborate with Edmond Schonberg on the writing of this book, which summarizes the experience achieved during these years. Currently his main line of research is the integration of the new Ada 2005 features into the GNAT compiler.

New York and the Canary Islands, June, 2004

Part I

<u>First Part: Introduction</u>

Chapter 1

The GNAT Project

GNAT is an acronym for GNU Ada Translator; a Front-End and Run-Time system for *Ada 95* that uses the GCC back-end as a retargettable code generator, and is distributed according to the guidelines of the *Free Software Foundation*. GNAT was initially developed by two cooperating teams:

- *New York University Team.* This group, led by Prof. Robert B.K. Dewar, and Edmond Schonberg was responsible for the development of the front-end of the compiler.
- *Florida State University Team.* This group, also known as the POSIX Ada Real-Time Team, was led by Prof. Theodore P. Baker, and was responsible for the first design of the concurrency components of the run-time library.

The NYU project was sponsored by the U.S. government from 1991 to 1994. In August, 1994 the members of the NYU team created the company *Ada Core Technologies, Inc.*, which provides technical support to industrial users of GNAT and has transformed GNAT into an industrial-strength, full-featured compiler: GNAT Pro). This compiler includes a modern tool suite and environment for the development of Ada-based software (i.e. GPS). Nowadays Ada Core continues investing resources to port GNAT to new architectures and operating systems, and has an active participation in the new revision of Ada (Ada 2005). Ada Core periodically makes available public versions of the compiler to the Ada community at large.

This chapter introduces the main components of GNAT. It is structured as follows: Section 1.1 briefly introduces GCC; Section 1.2 presents the main components of the GNAT compiler. Finally, Section 1.3 gives an overview of the GNAT compilation model.

1.1 GCC

GCC [Sta04] is the compiler system of the GNU environment. GNU (a self-referential acronym for 'GNU is Not Unix') is a Unix-compatible operating system, being developed by the Free Software Foundation, and distributed under the GNU Public License (GPL). GNU software is always distributed with its sources, and the GPL enjoins anyone who modifies GNU software and redistributes the modified product to supply the sources for the modifications as well. Thus, enhancements to the original software benefit the software community at large.

GCC is the centerpiece of the GNU software. It is a compiler system with multiple front-ends and a large number of hardware targets. Originally designed as a compiler for C, it now includes front-ends for C++, Objective-C, Ada, Fortran, Java, and treelang. Technically, the crucial asset of the GCC is its mostly language-independent and target-independent code generator, which produces excellent quality-code both for CISC and RISC machines. Remarkably, the machine dependences of the code generator represent less than 10% of the total code. To add a new target to GCC, an algebraic description of each machine instruction must be given using a Register-Transfer Language (RTL). Most of the code generation and optimization then uses the RTL, which GCC maps when necessary into the target machine language. Furthermore, GCC produces high-quality code, comparable to that of the best commercial compilers.

1.2 The GNAT Compiler

The first decision involved choosing the language in which GNAT compiler should be written. GCC is fully written in C, but for technical reasons, as well as non-technical ones, it was inconceivable to use anything but Ada for GNAT itself. In fact, the definition of the Ada language depends heavily on hierarchical libraries, and cannot be given except in Ada 95, so that it is natural for the compiler and the environment to use child units throughout.

The GNAT team started using a relatively small subset of Ada83, and in typical fashion, extended the subset whenever new features became implemented. Six months after the coding started in earnest, they were able to bootstrap the compiler, and abandon the commercial compiler they had been using up to that point. As soon as more Ada95 features were implemented, they were able to write GNAT in Ada95.

The GNAT compiler is composed of two main parts: the *Front-End* and the *Back-End* (cf. Figure 1.1). The front-end of is written in Ada 95, and the back-end is the GCC back-end extended to meet the needs of Ada semantics (i.e. exceptions support).

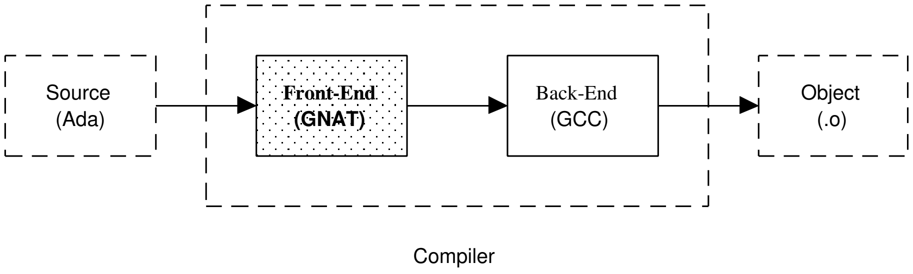


Figure 1.1: GNAT Compiler.

The front-end comprises five phases (cf. Figure 1.2): Lexical Analysis (Scanning), Syntax Analysis (parsing), Semantic Analysis, Expansion, and GIGI phases. The scanner analyzes the input characters and generates the associated *Tokens*. The parser verifies the syntax of the tokens and creates the Abstract Syntax Tree (AST). The semantic analyzer performs all static legality checks on the program and decorates the AST with semantic attributes. The expander transforms high-level AST nodes (nodes representing tasks, protected objects, etc.) into equivalent AST fragments built with lower-level abstraction nodes and, if required, calls to Ada Run-Time library routines. Given that code generation requires that such fragments carry all semantic attributes, every expansion activity must be followed by additional semantic processing on the generated tree (see the backward arrow from the expander to the semantic analyzer). At the end of this process the *GIGI* phase transforms the AST into a tree which is read by the GCC back-end (GNAT to GNU transformation phase). This phase is really an interface between the GNAT front-end and the GCC back-end. In order to bridge the semantic gap between Ada and C, several GCC code generation routines have been extended, and others added, so that the burden of translation is also assumed by GIGI and GCC whenever it is awkward or inefficient to perform the expansion in the front-end. For example, there are code generation actions for exceptions, variant parts and accesses to unconstrained types. As a matter of GCC policy, the code generator is extended only when the extension is likely to be of benefit to more than one language.

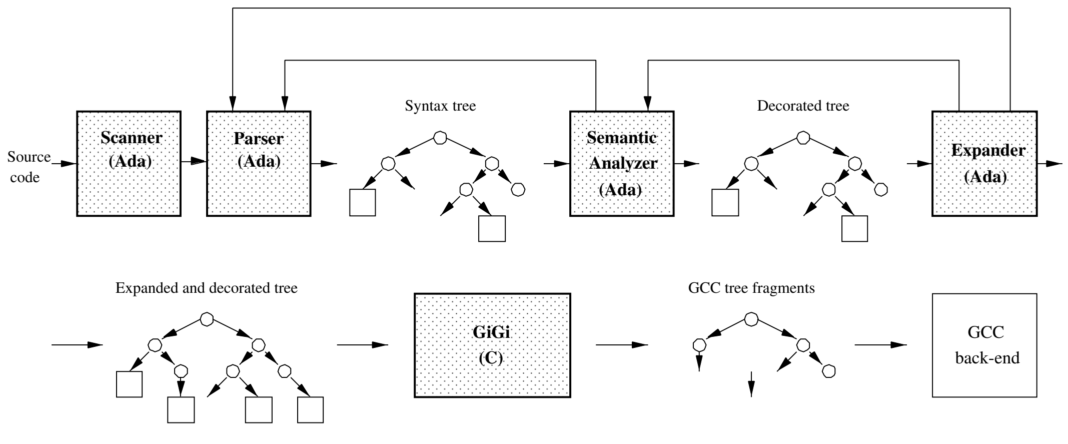


Figure 1.2: GNAT Front-End Phases.

All these phases communicate by means of a compact Abstract Syntax Tree (AST). The implementation details of the AST are hidden by several procedural interfaces that provide access to syntactic and semantic attributes. It is worth mentioning that strictly speaking GNAT does not use a symbol table. Rather, all semantic information concerning program entities is stored in defining occurrences of these entities directly in the AST.

There is a further unusual recursive aspect to the structure of GNAT. The program library (described in the next section) does not hold any intermediate representation of compiled units. As a result, if the expander generates a call to a Run-Time Library routine, the compiler requires the specification of the corresponding Run-Time package to be analyzed as well (see the backward arrow from the expander to the parser).

1.3 Compilation Model

The notion of program library is one of the fundamental contributions of Ada to software engineering. The library guarantees that type safety is maintained across compilations, and prevents the construction of inconsistent systems by excluding obsolete units. In most Ada compilers, the library is a complex structure that holds intermediate representations of compiled units, information about dependences between compiled units, symbol tables, etc. GNAT has chosen a different approach: the separate files that constitute the program are separately compiled, and each compilation produces a corresponding object file. These object files are then linked together by specifying a list of object files in a program. Thus, the

Ada library consists of a set of such object files (there is no library file as such). In the following sections we briefly present both alternatives.

1.3.1 Traditional Compilation Model

In the traditional model, an Ada library is a data structure that gathers the results of a set of compilations of Ada source files. A compilation is performed in the context of such a library, and the information in the library is used to enforce type consistency between separately compiled modules. Unlike some other language environments, all such type checking is performed at compile time, and Ada guarantees at the language level that separately compiled modules of a complete Ada program are type consistent.

In this model, building an Ada program consists of selecting a main program (a parameterless procedure compiled into the Ada library), and all the modules on which this main program depends, and bound them into a single executable program. A definite order of compilation is enforced by the language semantics and implemented by means of the Ada library. Basically, before a compilation unit is compiled, the specification of all the units on which it depends must be compiled first. This gives the Ada compiler a fair amount of freedom in the compilation order. An important consequence of this model is the notion of *obsolete* unit. If a unit is recompiled, then units which depend on it become obsolete, and must be recompiled. Again, the Ada library is the data structure used to implement this requirement.

In the Ada Reference Manual [AAR95, Chapter 10], there are specific references to a *Library File*, and this is often taken to mean that the Ada Library should be represented using a file in the normal sense. Most Ada systems do in fact implement the Ada library in this manner. However, it is generally recognized that the Ada Reference Manual does not require this implementation approach. In this view, an Ada library is a conceptual entity that can be implemented in any manner that supports the required semantics. In fact the monolithic library approach is ill-adapted to multi-language systems, and has been responsible for some of the awkwardness of interfacing Ada to other languages.

1.3.2 GNAT Compilation Model

GNAT has chosen a completely different approach: sources are independently compiled to produce a set of objects, and the set of object files thus produced is submitted to the binder/linker to generate the resulting executable (cf. Figure 1.3). This approach removes all order of compilation considerations, and eliminates the traditional monolithic library structure. The library itself is implicit, and object files depend only on the sources used to compile them, and not on other objects. There are no intermediate representations of compiled units, so that unit declarations appearing in context clauses of a given compilation are always analyzed anew. Dependency information is kept directly in the object files (in fact, they are kept in a small separate file, conceptually linked to the object file), and amounts to a few hundred bytes per unit.

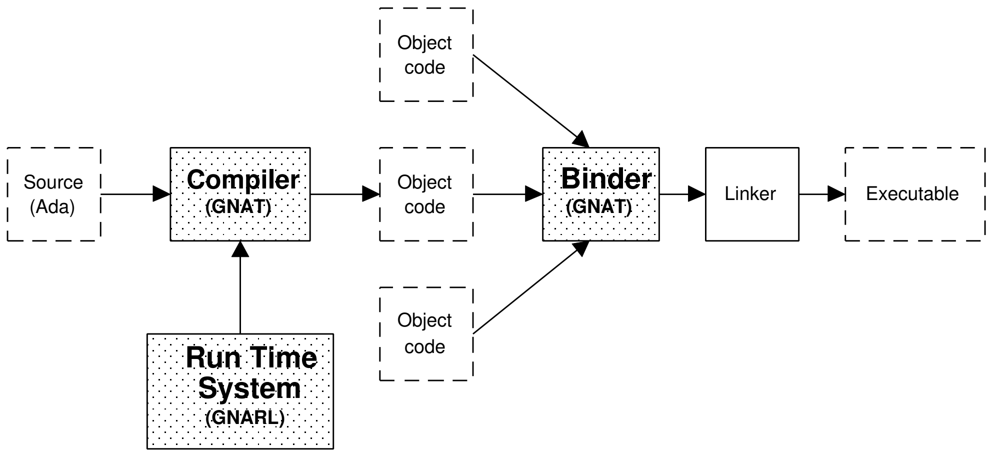


Figure 1.3: GNAT Overall Structure.

Given the speed of the GNAT front-end, this approach is no less efficient than the conventional library mechanism, and has the following advantages over it:

1. The compilation of an Ada unit is identical to the compilation of a module or file in another language: the result of the compilation of one source is one object file.

2. Given that inlining is always done from the source, there is no requirement that the entities to be inlined should be compiled first. It is even possible for two bodies to inline functions defined in each other, without fear of circularities. Thus inlining works in a much more flexible way than in normal Ada compilers.

3. The standard system utilities to copy, rename and remove files can be reused to copy, rename and remove object modules.

4. Since GNAT uses the same compilation model as other languages, it is also much easier to build programs where various parts of the program are built

in different languages. Furthermore, GCC is committed to common system standard conventions for calling sequences, object module formats, including debugging information, and data structure layouts, so it is also easy to integrate Ada with any language supported by GCC. GNAT even makes possible to write multi-language programs whose main program is not itself written in Ada.

5. It is more compatible with conventional configuration management tools than the conventional library structure (tools ranging from the simple UNIX *make* program to sophisticated compilation management environments).

In the GNAT model, a source file contains a single compilation unit, and a compilation is represented as a series of source files, each of which contains one compilation unit. Furthermore there is a direct mapping from unit names to file names, so that from a unit name one can always determine the name of the file that contains the source for that unit. The default file naming convention is as follows: (1) The file name is the expanded name of the unit, with dots replaced by minus sign, (2) The extension ".ads" is used for specifications, and the extension ".adb" for bodies. Only the body produces an object file, so the fact that the specification and body have the same file name does not cause difficulties. The object file conceptually contains the Ada Library Information for that source (extension ".ali") whose most important component is a recording of the time stamps of the compilation units on which a compiled unit depends.

In this model the compilation of a source file may require other source files. These include:

1. The corresponding specification for a body.
2. The parent specification of a child library spec.
3. Specifications of with'ed units.
4. Parent body for a subunit.
5. Bodies of inlined subprograms.
6. Bodies of instantiated generics.

The key understanding is that in GNAT, dependencies are not established from one compilation unit to another, but from object files to corresponding source files. In this context GNAT is re-interpreting the Ada "order of compilation" rules to be "dependency on source files" rules. The rules regarding compilations that obsolete other compilations are similarly reinterpreted. For example, a rule that says: *The body of package cannot be compiled until its specification has been compiled*, is re-interpreted to mean: *The body of package cannot be compiled unless the source of its specification is available*. One interesting consequence of this approach is that if all the sources of a program are available, there are in fact no restrictions on the order of compilation. This feature facilitates the parallel compilation of Ada programs.

The main argument against the GNAT model is that the compiler is constantly recompiling the specification of with'ed units. However, the alternative is not better. In traditional Ada library-based systems, the result of a compilation is to place information, typically some kind of intermediate tree, in the library. A subsequent with_clause then fetches this tree from the library. In practice, this tree information can be huge, often much bigger than the source. Furthermore, it is generally a complex interlinked data structure. Thus it is not clear that re-reading and recompiling the source is less efficient than writing and reading back in these trees. It's true that recompiling means redoing syntax and semantic checking, but this causes less Input/Output than reading and writing linked structures. On the contrary, the GNAT model gives all the previously discussed advantages.

The Binder

Ada establishes the rules which determine valid orders of elaboration [AAR95, Section 10.2]. It is also possible to construct programs for which no possible order of elaboration exists. Such programs are illegal, and must be diagnosed prior to execution. Because this work can not be established until all the object files are available, GNAT needs an special pre-linker (the *binder*) which establishes a valid sequence of calls to the initialization procedures for specifications and bodies (cf. Figure 1.3).

Part of the processing in the GNAT binder ascertains that the program is consistent by looking at time stamps in the ALI files associated with the compilation units required for the program. The binder consistency checks can be done in one of three modes:

1. From ALI files only.
2. From ALI files and any corresponding sources that can be located.
3. From ALI files and all corresponding sources, which must be available.

Despite the clear advantages of operating in "source file" mode (second and third alternative), it is more useful for the GNAT binder to operate in "ali files only" mode. Not only is this mode faster, since no source files need to be accessed, but more importantly, it means that GNAT programs can be linked from objects even if the sources are not available. This is indispensable when linking libraries that for proprietary reasons may be distributed without the sources for their bodies. Therefore it is the mode implemented in the GNAT Binder.

1.4 Summary

This introductory chapter has presented the overall structure of the GNAT project. The compiler has two main parts: the *front-end* and the *back-end*. The *front-end* comprises five phases which communicate by means of an *Abstract Syntax Tree*. The *back-end* is the GCC target independent code generator, what gives two main advantages: portability and excellent-quality code generation.

The most novel aspect of the GNAT architecture is the source-based organization of the library. In most Ada compilers the library is a monolithic complex structure that holds intermediate representations of compiled units. GNAT library model follows the traditional model used by nearly all languages throughout the entire history of programming languages: there is no centralized library, a source file contains a single compilation unit, and a compilation specifies a source file and generates a single object file. This model is fully conformant with the prescribed semantics given in the Ada Reference Manual and, at the same time, enables the use of many well-known configuration management tools (i.e. UNIX make), simplifies the construction of multi-language programs, and allows the parallel compilation of the Ada programs. Because the Ada language gives the rules which govern the order of elaboration of the compilation units the GNAT model needs a special pre-linker (the binder) which verifies the object files and generates a valid order of elaboration.

Chapter 2

Overview of the Front-end Architecture

The GNAT front-end comprises four phases that communicate by means of a compact *Abstract Syntax Tree* (AST): lexical analysis, syntax analysis, semantic analysis, and expansion. This chapter provides an overview of the architecture of these phases. It is structured as follows: Section 2.1 presents the scanner architecture; Section 2.2 gives an overview of the parser, describes the high-level specification of the AST nodes and presents the mechanisms used to resynchronize it in case of syntax errors; Section 2.3 describes the architecture of the semantic analyzer, and finally Section 2.4 discusses the architecture of the expander.

2.1 The Scanner

For efficiency reasons no automatic tool was used to generate the GNAT scanner. It is a subprogram of the parser that reads input characters, identifies the next token, and returns it to the parser. To give support to various operating systems and to multiple languages, it is engineered to handle multiple character encodings (cf. Package *Csets*). Figure 2.1 presents its architecture: package *Scn* has most of the implementation of the scanner; package *Scans* contains the *Tokens* definition and the state of the automaton. Finally, package *Snames* has the standard names (Ada keywords, pragmas and attributes). Low level package *Namet* handles name storage and look up, and package *Opt* has the global flags set by command-line switches and referenced throughout the compiler.

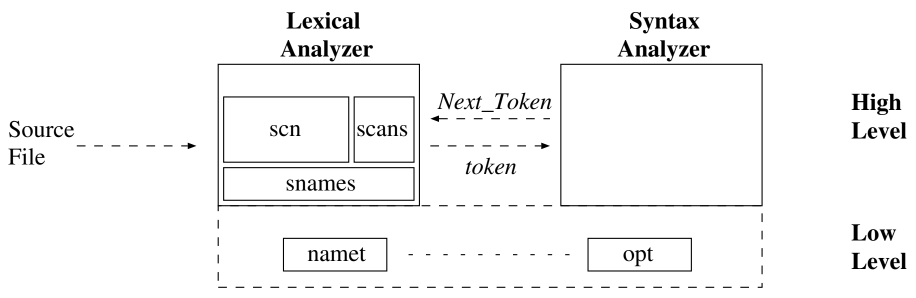


Figure 2.1: Architecture of the GNAT Scanner

2.2 The Parser

The GNAT parser is a hand-coded recursive descent parser. The main reasons which justify this choice (rather than the traditional academic choice of a table-driven parser generated by a tool) were [CGS94, Section 3.2]:

- **Better error messages**. GNAT generates clear and precise error messages. Even in case of serious structural errors, including the interchange of ";" and "is" between specification and body of a subprogram, the parser posts a precise and intelligible message. Bottom-up parsers have serious difficulties with such errors.

- **Clarity**. The GNAT parser follows faithfully the grammar given in the Ada Reference Manual [AAR95]. This has clear pedagogical advantages because the parser can be easily read in conjunction with the ARM, and makes it easier to maintain, for example to add new error recovery techniques. The Ada grammar given the ARM is ambiguous, and a table-driven parser would be forced to modify the grammar to make it acceptable to LL (1) or LALR (1) techniques. No such problem arises for a recursive descent parser, because when necessary the parser can perform arbitrary look-ahead.

- **Performance**. The GNAT parser is as fast as any Ada table driven parser, and arguably faster than a LALR parser.

In most of the cases the parser is guided by the next token provided by the scanner. However, when dealing with ambiguous syntax rules the parser saves the state of the scanner, and issues repeated calls to look-ahead the following tokens and thus resolve the ambiguity (cf. *Save_Scan_State* and *Restore_Scan_State* in package *Scans*).

In addition to syntax verification, the GNAT parser builds the *Abstract Syntax Tree* (AST), that is to say the structured representation of the source program. This tree is subsequently used by the semantic analyzer to perform all the static checks on the program, that is to say all the context-sensitive legality rules of the language.

At the architectural level the main subprogram of the GNAT parser is an Ada function (*Par*) that is called to analyze each compilation unit. The parser code is organized as a set of packages (subunits of the top-level function) each of which contains the parsing routines associated with one chapter of the Ada Reference Manual [AAR95]. For example, package *Par.Ch2* contains all the parsing subprograms associated with the second chapter of the Ada Reference Manual (cf. Figure 2.2). In addition, the name of the parsing subprograms follow a fixed rule: Prefix "*P_*" followed by the name of the corresponding Ada syntax rule (for example, *P_Compilation_Unit*).

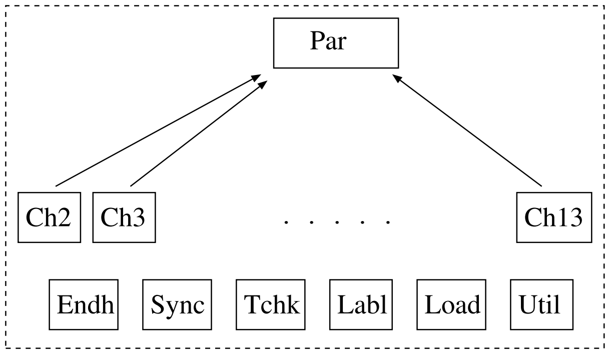


Figure 2.2: Structure of the GNAT Parser

The GNAT Parser also has several additional sub-units: Package *Endh*, contains subprograms to analyze the finalization of the syntax scopes; package *Sync*, contains subprograms to resynchronize the parser after syntax errors (cf. Chapter 3); package *Tchk*, contains subprograms that simplify the verification of tokens; procedure *Labl* handles implicit label declarations; procedure *Load* controls the loading into main memory of successive compilation units; function *Prag* analyzes *pragmas* that affect with the behaviour of the parser (such as lexical style checks), and finally package *Util* has general purpose subprograms used by all the parsing routines.

Each parsing routine carries out two main tasks: (1) Verify that a portion of the source obeys the syntax of one particular rule of the language, and (2) Build the corresponding Abstract Syntax Subtree (cf. Figure 2.3).

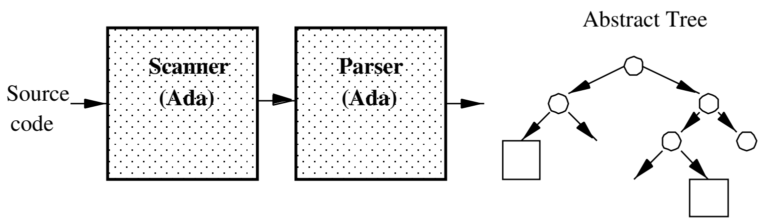


Figure 2.3: Abstract Syntax Tree Construction.

2.2.1 The Abstract Syntax Tree

The GNAT Abstract Syntax Tree (AST) has two kind of nodes: internal (structural) nodes that represent the syntactic structure of the progam, and extended nodes that store information about Ada entities (identifiers, operator symbols and character literal declarations). Internal nodes have 5 general purpose fields which can be used to reference other nodes, lists of nodes (i.e. the list of statements in an Ada block), names, literals, universal integers, floats, or character codes. Entity nodes have 23 general purpose fields, and a large number of boolean flags, that are used to store in the tree all relevant semantic attributes of each entity. In other compilers this information is commonly stored in a separate symbol table.

Figure 2.4 describes the GNAT packages involved in the AST handling. Low level package *Atree* implements and abstract data type that contains the definitions related with structure nodes and entities, as well as subprograms to create, copy, and delete nodes, and subprograms to read and modify the general purpose fields. Low level package *Nlists* provides the support for handling lists of nodes. Packages *Sinfo* and *Einfo* contain the high-level specification of the nodes, that is, the high-level names associated with the low-level general purpose node fields, and subprograms to read and modify these fields with their high-level names. Package *Nmake* has subprograms to create high level nodes with syntax and semantic information.

Let us examine the format of the high-level specification of the nodes by means of an example. The Ada syntax rule for a package body is:

```
PACKAGE_BODY ::=
   package body DEFINING_PROGRAM_UNIT_NAME is
      DECLARATIVE_PART
   [begin
      HANDLED_SEQUENCE_OF_STATEMENTS]
   end [[PARENT_UNIT_NAME .] IDENTIFIER];
```

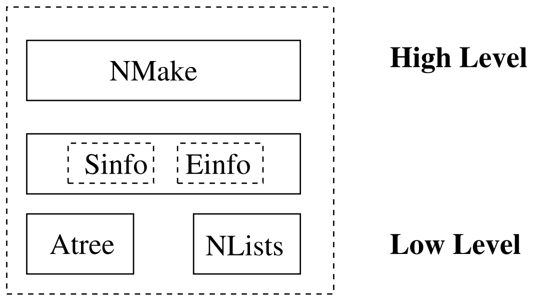


Figure 2.4: Abstract Syntax Tree Packages.

The corresponding high-level node is specified in the package *Sinfo* as follows:

```
--  N_Package_Body
--  Sloc points to PACKAGE
--  Defining_Unit_Name (Node1)
--  Declarations (List2)
--  Handled_Statement_Sequence (Node4) (set to Empty
--     if not present)
--  Corresponding_Spec (Node5-Sem)
--  Was_Originally_Stub (Flag13-Sem)
```

The first line specifies the node kind (*N_Package_Body*), which is an enumerated literal of type *Sinfo.Node_Kind*; the second line indicates that the source coordinate (Sloc) for the node is the source coordinate of the keyword that is the first token in the production. This source coordinate is used whenever errors or warnings are posted on a given construct. The following lines specify the high-level names given to the general purpose fields. Their format is: (1) High-level Name of the field (chosen for its syntactic significance) and, (2) Data type, and placement of the corresponding low-level general-purpose field (this information is enclosed in parenthesis). In addition some fields may specify a default initialization value For example, the field named *Handled_Statement_Sequence* references the node that represents the statements-sequence and the exception handlers found in the optional package initialization. This field is placed in the fourth general-purpose low-level field of a node, and is set to the value *Empty* if the package body has no initialization code. To handle uniformly the AST, identical node fields associated with different nodes are always assigned to the same low-level general purpose fields. The last two lines specify two semantic attributes (indicated by the suffix *"-Sem"*). Semantic attributes are computed and annotated into the tree by the Semantic Analyzer in the next phase of the compiler (cf. Section 2.3). Figure 2.5 represents the subtree associated with this high-level node specification.

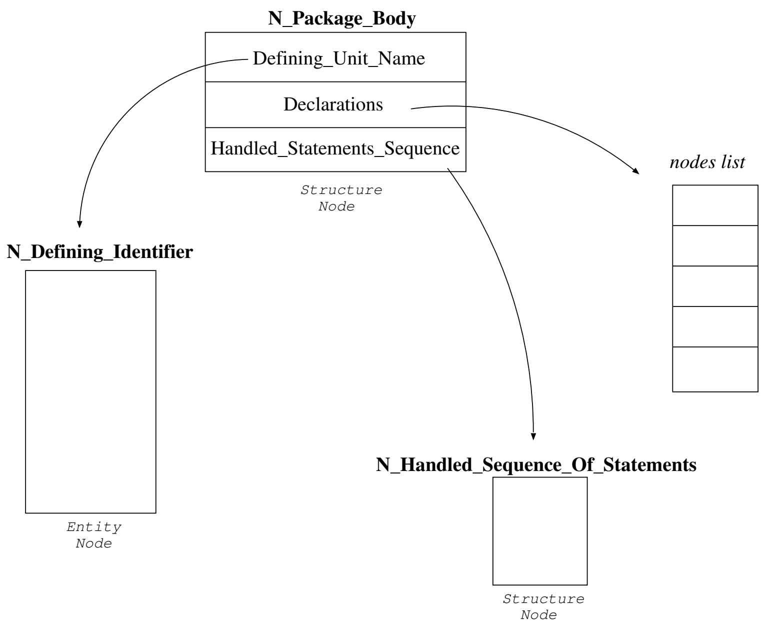


Figure 2.5: Abstract Syntax Subtree associated with the package body rule.

The high-level specification of the AST nodes is read by the GNAT tools *xsinfo*, *xtreeprs* and *xnmake*, which generate some complementary Ada packages of the front-end that use the low-level package *Atree* to provide the specified functionality.

2.3 The Semantic Analyzer

The GNAT Semantics Analyzer traverses the Abstract Syntax Tree built by the parser, verifies the static semantics of the program, and decorates the AST, that is to say adds the semantic information to the AST nodes (cf. Figure 2.6).

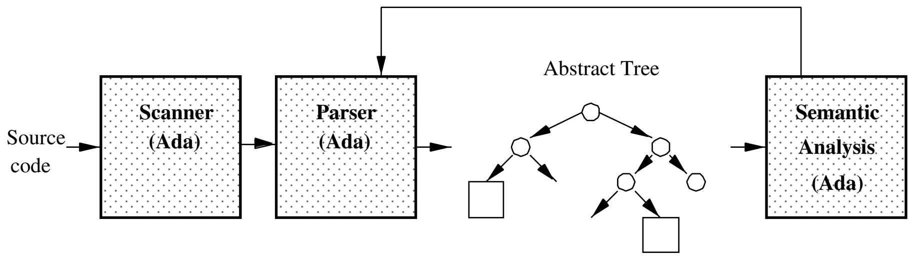


Figure 2.6: Abstract Syntax Tree Decoration.

In general the static analysis carried out by the compiler implies the following tasks: (1) Group entities in scopes and resolve referenced names, 2) Handle private types, 3) Handle discriminants, 4) Analyze and instantiate generic units, and 5) Handle freezing of entities. These topics will be discussed in detail in the following chapters.

Figure 2.7 presents the architecture of the GNAT Semantic Analyzer. The overall structure is similar to that of the parser, that is to say it parallels the organization of the ARM. For example, Sem_Ch3 deals handling of types and declarations, Sem_Ch9 with concurrency, and Sem_Ch12 with generics and instantiations. The name of individual semantic analysis subprogram follow a fixed rule: they have the prefix "Analyze_" and a suffix that names the analyzed language construct. ie. Analyze_Compilation_Unit. Exceptions to this general rule are packages *Sem_Prag* and *Sem_Attr* which correspond to language elements that are described throughout the ARM, and which also constitute a basic extension mechanism for the compiler.

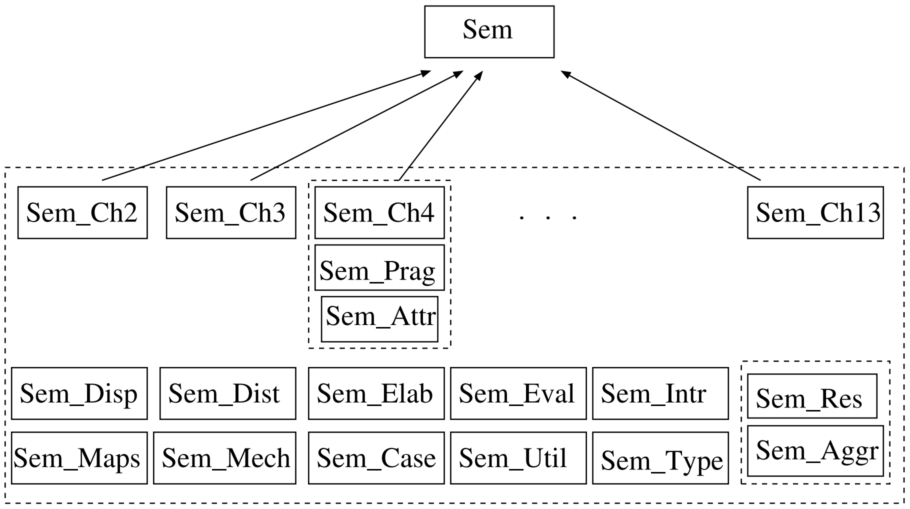


Figure 2.7: Structure of the Semantic Analyzer.

In addition, the GNAT Semantic Analyzer has some utility packages for specialized purposes. *Sem_Disp* has routines involved in the analysis of tagged types and dynamic dispatching; *Sem_Dist* contains subprograms which analyze the Ada Distributed Systems Annex [AAR95, Annex E]; *Sem_Elab* contains the routines that deal the order of elaboration of a set of compilation units; *Sem_Eval* contains subprograms involved in compile-time evaluation of static expressions and legality checks for staticness of expressions and types. *Sem_Intr* analyzes the declaration of intrinsic operations; *Sem_Mech* has a single routine that analyzes the declaration of calling mechanisms for subprograms (needed for the VMS version of GNAT). *Sem_Case* has routines to process lists of discrete choices (such lists can occur in 3 different constructs: case statements, array aggregates and record variants); package *Sem_Util* has general purpose routines used by all the semantics packages. Finally package *Sem_Type* has subprograms to handle sets of overloaded entities, and package *Sem_Res* has the implementation of the well-known two-pass algorithm that resolve overloaded entities (described in Chapter 5). Package *Sem_Aggr* is conceptually an extension of *Sem_Res*; it has been placed in a separate package because of the complexity of the code that handles aggregates.

The main package (*Sem*) implements a dispatcher which receives one AST Node and calls the corresponding analysis subprogram (cf. Figure 2.8). Called routines recursively call *Analyze* to do a top-down traversal of the tree.

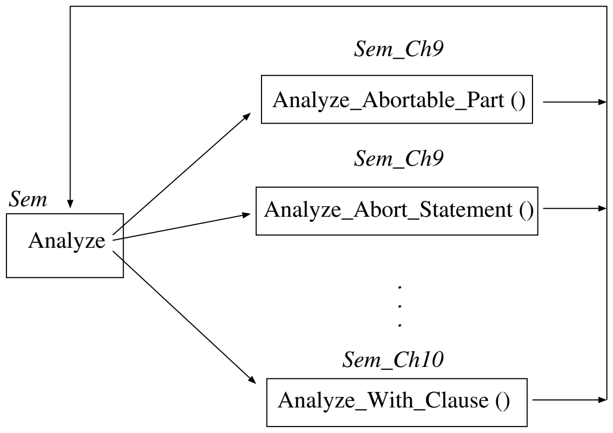


Figure 2.8: Abstract Syntax Tree Nodes Dispatching.

The resolution routines are called by the analysis routines to resolve ambiguous nodes or overloaded entities. For example, the Ada syntax of a procedure-call statement is exactly the same as that of an entry-call statement. Given this ambiguous syntactic specification, the GNAT parser generates the same *N_Procedure_Call* node for both cases, and the Semantic Analyzer must analyze the context, determine the nature of the entity being called, and, when needed replace the original node by an entry-call statement node (which will be subject to completely different expansion that a procedure call). To resolve overloaded entities GNAT implements a well-known two-pass algorithm. During the first (bottom-up) pass, it collects the set of possible meanings of a name. In the second pass, the type imposed by the context is used to resolve ambiguities and chose a unique meaning for each overloaded identifier in the expression. This is described in detail in Chapter 5.

2.4 The Expander

The GNAT expander performs AST transformations for those constructs that do not have a close equivalent in the C-level semantics of the back-end. (cf. Figure 2.9). Its main expansions are [CGS94, Section 3.3]:

- Replace nodes which represent high-level Ada constructs by nodes with represent lower-level abstractions. For example, the node which represents an Ada task body is replaced by one node that represents a procedure, plus one call to the Run-Time Library subprogram which creates the corresponding thread of control (cf. Chapters 9 to 13).
- Replace nodes that represent a generic instantiation by a copy of the corresponding AST, with appropriate substitutions (cf. Chapter 7).
- Build *Type Support Subprograms* (TSS), which are internally generated subprograms associated with particular types. For example, implicit initialization subprograms, and subprograms to implement Ada *Input/Output* attributes (cf. Package *Exp_TSS*).

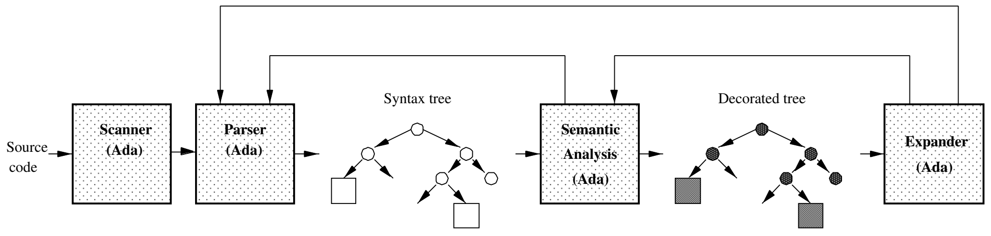


Figure 2.9: Abstract Syntax Tree Expansion.

Given that code generation requires every AST node carry all needed semantic attributes, every expansion activity must be followed by additional semantic processing on the generated fragments (see backward arrows in Figure 2.9). As a result the two phases (analysis and expansion) are recursively integrated into a single AST traversal. After the whole AST is analyzed and expanded, the resulting AST is passed to Gigi in order to generate the GCC tree-fragments that are the inputs to the back-end code generator.

The architecture of the GNAT Expander follows the same scheme of the previous phases: the expansion subprograms are grouped into packages following the Ada Reference Manual chapters (cf. Figure 2.10), and the name of the subprograms follow a fixed rule: Prefix “Expand_” followed by the corresponding rule of the language (ie. Expand_Compilation_Unit). Similar to the Semantic Analyzer, the package *Expander* implements a dispatcher which receives one node and calls the corresponding expander subprogram.

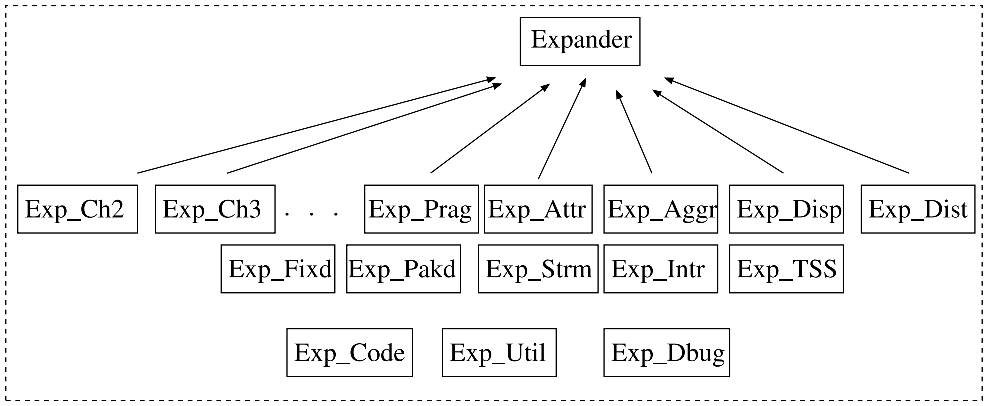


Figure 2.10: Architecture of the GNAT Expander

The GNAT expander has the following packages: *Exp_Prag*, that groups routines to expand pragmas; *Exp_Attr* has subprograms to expand Ada attributes; *Exp_Aggr* contains subprograms to expand Ada aggregates; *Exp_Disp*, with routines involved in tagged types and dynamic dispatching expansion; *Exp_Dist* has routines used to generate the stubs of the Ada Distribution Annex [AAR95]; *Exp_Fixd*, subprograms to expand fixed-point data-type operations; *Exp_Pakd*, routines to expand packed arrays; *Exp_Strm*, routines to build stream subprograms for composite types (arrays and records); *Exp_TSS*, routines for Type Support Subprogram (TSS) handling; *Exp_Code*, subprograms to expand the Ada *code* statement [AAR95, Section 13.8], used to add machine code instructions to Ada source programs; *Exp_Util*, utility subprograms shared by expansion subprograms; and finally *Exp_Dbug* is a package with routines that generate special declarations used by the debugger.

2.5 Summary

The GNAT scanner implements a deterministic automaton which is called by the parser to get the next *token*. The GNAT Parser is a recursive-descent parser which not only verifies the syntax of the sources but also generates the corresponding Abstract Syntax Tree (AST). The AST has two kinds of nodes: *structure nodes* which represent the program structure, and *entity nodes* which store the information of Ada entities. Therefore GNAT has no separate symbol table; all the information traditionally stored in this table is kept in the AST entities.

The Semantic Analysis phase carries out a top-down traversal of the AST to do the static analysis of the program and decorate the AST. This phase implements a well-known two pass algorithm to resolve overloaded entities. The Expansion phase replaces high-level nodes by sub-trees with low-level nodes which provide the equivalent semantics and can be handled by the GCC code generator.

To help reading the sources the architecture of the parser, semantics and expander follow a fixed scheme: the subprograms are grouped into packages following as reference the Ada Reference Manual, and their names follow a fixed rule: one fixed prefix plus the corresponding rule of the Ada Reference Manual. The main package of the semantics and the expander implement a dispatcher which receives as input an AST node and calls the corresponding processing routine.

Chapter 3

Error Recovery

The GNAT scanner implements some basic error recovery techniques which simplify the implementation of the parser. The GNAT parser has what is considered to be the best set of error recovery strategies of any Ada compiler in use. In this chapter we briefly present these strategies. It is structured as follows: Section 3.1 introduces the error recovery techniques implemented in the scanner, and Section 3.2 presents the mechanisms used to resynchronize the parser. Section 3.2.1 presents the parser scope-stack which is used to handle nested scopes parsing; Section 3.1.1 discusses the use of the programmer casing convention to distinguish keywords from user-defined identifiers, and finally Sections 3.2.3 and 3.2.4 discuss the special handling of the keyword 'is' used in place of semicolon.

3.1 Scanner Error Recovery

The GNAT scanner implements a simple error recovery technique which simplifies error handing to the parser. After detecting a lexical error, the scanner posts the corresponding error message and returns one heuristic Token which masks the error to the next phases of the compiler. For example:

- If the character "[" is found, it assumes that the intention of the programmer was to write "(" (which is the right character used in Ada). Therefore, it posts the corresponding error message and returns the "left parenthesis" token to the parser.
- If the scanner detects the invalid Ada character sequence valid in other standard language, it assumes that the programmer intention was to use the

corresponding equivalence (if any) in Ada. For example, in front of the sequence "`&&`" (which is the syntax of the C *and* operator), it assumes that the programmer intention was to use the Ada *and* operator. Thus it posts the precise error message *"&& should be 'and then' "* and returns the "and then" token to the Parser. The scanner has many error messages specific for C programmers.

In most cases this simple but powerful mechanism helps masking lexical errors to the parser. This simplifies the implementation of the parser, which does not need to repeatedly handle them in many contexts. For additional details, read the *Scan*, *Nlit*, and *Slit* subprograms in package *Scn*.

3.1.1 Use of the Casing of Identifiers

The GNAT scanner assumes that the user has some consistent policy regarding the casing convention used to distinguish keywords from user-defined entities. The scanner deduces this convention from the first keyword and identifier that it encounters (cf. Package *Casing*). In the following example GNAT makes use of the upper/lower case rule for identifiers to treat the word **exception** as an indented identifier rather than the beginning of an exception handler (cf. subprogram *scan_reserved_identifier*).

```
procedure Wrong1 is
   Exception : Integer;
   |
   >>> reserved word "exception" cannot be used as identifier
begin
  null;
end Wrong1;
```

3.2 Parser Error Recovery

The GNAT parser includes a sophisticated error recovery system which, among other things, takes indentation into account when attempting to correct scope errors. When an error is encountered, a call is made to one parser routine to record the error (cf. Package *Errout*). If the parser can recover locally, it masks the failure to the next stages of the front-end by generating the AST nodes as if the syntax were right, and the parsing continues unimpeded. On the other hand, if the error represents a situation from which the parser cannot recover locally, the exception *Error_Resync*) is raised after the call to the routine that records the error. Exception handlers are located at strategic points to resynchronize the parser. For example, when an error occurs in a statement, the handler skips to the next semicolon and continues the scan from there.

In the GNAT sources, each parsing routine has a note with the heading "Error recovery" which shows if it can propagate the Error_Resync exception (cf. Files par-ch2.adb to par-ch13.adb). To avoid propagating the exception, a procedure must either contain its own handler for this exception, or it must not call any other routines which propagate the exception.

3.2.1 The Parser Scope-Stack

Many rules of the language define a syntax scope (rules ended with 'end'). For example, the syntax rules of packages, subprograms and all the flow-of-control statements. The GNAT parser uses a *scope-stack* to record the scope context. An entry is made when the parser encounters the opening of a nested construct, and then package *Endh* uses this stack to deal with 'end' lines (including properly dealing with 'end' nesting errors).

In case of parsing resynchronization by means of the exception mechanism (cf. Section 3.2), the arrangement of the exception handlers is such that it should never be possible to transfer control through a procedure which made an entry in the scope stack, invalidating the contents of the stack.

In some cases, at the end of a syntax scope, the programmer is allowed to specify a name (ie. at the end of a subprogram body); other scope rules have a rigid format (ie. at the end of a record-type definition). In the first case, it is a semantic error to open a syntax scope with a name and to close it with a different name. Although many Ada compilers detect this error in the phase of semantic analysis, GNAT uses the parser scope-stack to detect it as soon as possible and thus simplify the semantics.

3.2.2 Example 1: Use of indentation

The next example combines the use of the scope-stack plus the indentation to match the statement:

```
procedure Wrong2 is
   A, B : Integer;
begin
  if A > B then
     null;
end Wrong2;
|
>>>"end if;" expected for "if" at line 4
```

Note that a more conventional approach to error recovery would have produced two errors: it would have identified the **end** with the if-statement, complained about a missing “if”, and then complained about the missing end for the procedure itself.

3.2.3 Example 2: Handling Semicolon Used in Place of 'is'

The two contexts in which a semicolon may have been erroneously used in place of 'is' are at the outer level, and within a declarative region. The first case corresponds to the following example:

```
-- Case 1: At the outer level
procedure X (Y : Integer);
   Q : Integer;
begin
    ...
end;
```

In this case the GNAT parser knows that something is wrong as soon as it encounters Q (or 'begin', if there are no declarations), and it can immediately diagnose that the semicolon should have been 'is'. The situation in a declarative region is more complex, and corresponds to the following example:

```
-- Case 2: Within a declarative region
declare
   procedure X (Y : Integer);   --<1>
      Q : Integer;
   begin                        --<2>
      ...
   end;
begin
   ...
end;
```

In this case, the syntax error (line <1>) has the syntax of a subprogram declaration [AAR95, Section 6-1]. Therefore, the declaration of Q is read as belonging to the outer region. The parser does not know that it was an error until it encounters the 'begin' (line <2>). It is still not clear at this point from a syntactic point of view that something is wrong, because the 'begin' could belong to the enclosing syntax scope. However, the GNAT parser incorporates a bit of semantic knowledge and note that the body of X is missing, so it diagnoses the error as semicolon in place of 'is' on the subprogram line. To control this analysis, some global variables with prefix "SIS_" are used to indicate that we have a subprogram declaration whose body is required and has not yet been found. For example, the variable *SIS_Entry_Active* indicates that a subprogram declaration has been encountered, but no body for this subprogram has been encountered yet. The prefix stands for "Subprogram IS" handling. Five things can happen to an active SIS entry:

1. If a 'begin' is encountered with an SIS entry active, then we have exactly the situation in which we know the body of the subprogram is missing. After posting an error message, the parser rebuilds the AST: it changes the specification to a body, re-chaining the declarations found between the specification and the word 'begin'.

2. Another subprogram declaration or body is encountered. In this case the entry gets overwritten with the information for the new subprogram declaration. Therefore, the GNAT parser does not catch some nested cases, but it does not seem worth the effort.

3. A nested declarative region (e.g. package declaration or package body) is encountered. The SIS active indication is reset at the start of such a nested region. Again, similar to the previous case, the parser misses some nested cases, but it does not seem worth the effort to stack and unstack the SIS information.

4. A valid pragma 'interface' or 'import' supplies the missing body. In this case the parser resets the entry.
5. The parser encounters the end of the declarative region without encountering a 'begin' first. In this situation the parser simply resets the entry: there is a missing body, but it seems more reasonable to let the later semantic checking discover this.

3.2.4 Example 3: Handling 'is' Used in Place of Semicolon

This is a somewhat trickier situation, and although the GNAT parser can not catch it in all cases, it does its best to detect common situations resulting from a "cut and paste" operation which forgets to change the 'is' to semicolon. Let us consider the following example:

```
package body X is
   procedure A;
   procedure B is       -- <1>
   procedure C;
   ...
   procedure D is
   begin
      ...
   end;
begin
   ...
end;                    -- <2>
```

The trouble is that the section of text from line `<1>` to line `<2>` syntactically constitutes a valid procedure body, and the danger is that the parser finds out far too late that something is wrong (indeed most compilers will behave uncomfortably on the above example).

To control this situation the GNAT parser avoids swallowing the last 'end' if it can be sure that some error will result from doing so. In particular, the parser will not accept the 'end' it if it is immediately followed by end of file, 'with' or 'separate' (all tokens that signal the start of a compilation unit, and which therefore allow us to reserve the 'end' for the outer level). For more details on this aspect of the handling, see package *Endh*. Similarly, if the enclosing package has no 'begin', then the result is a missing 'begin' message, which refers back to the subprogram header. Such an error message is not too bad (it's already a big improvement over what many parsers do), but it's not ideal, because the declarations following the 'is' have been associated with the wrong scope.

To catch at least some of these cases, the GNAT parser carries out the following additional steps. First, a subprogram body is marked as having a suspicious 'is' if the declaration line is followed by a line which starts with a symbol that can start a declaration in the same column, or to the left of the column in which the 'function' or 'procedure' starts (normal style is to indent any declarations which really belong a subprogram). If such a subprogram encounters a missing 'begin' or missing 'end', then the parser decides that the 'is' should have been a semicolon, and the subprogram body node is marked (by setting the Bad_Is_Detected flag to true). This is not done for library level procedures since they must have a body.

The processing for a declarative part checks to see if the last declaration scanned is marked in this way, and if it is, the tree is modified to reflect the 'is' being interpreted as a semicolon.

3.3 Summary

GNAT implements an heuristic error recovery mechanism on it which simplifies the implementation of the parser: when an error is detected, the scanner generates the corresponding error message, estimates a substitute token and returns it to the parser. In most cases this simple, but powerful, mechanism helps the parser to continue as if the source program had no lexical errors.

In case of simple errors the parser masks the failure and generates the right nodes as if the source program were correct. In case of complex errors, the parser implements a resynchronization mechanism based on exception handlers.

Part II

Second Part: Semantic Analysis

Chapter 4

Scopes and Visibility

Modern high-level programming languages allow us to define names with a limited scope of visibility, to reuse them in different contexts and also to overload them; It is responsibility of the compiler to keep track of scope and binding information about names. For this purpose the compilers generally use a separate *Symbol Table* structure. In the case of GNAT, there is no separate symbol table. Rather, semantic information concerning program entities is stored directly in the Abstract Syntax Tree (AST). The GNAT AST is thus close in spirit to DIANA [GWEB83], albeit more compact.

This chapter is structured as follows: Section 4.1 presents the entity flags and data structures used by the front-end to handle scopes and visibility; Section 4.2 describes the sequence of steps followed to analyze records, tasks, protected objects, context clauses, child units and subunits, and finally Section 4.5 describes the sequence of steps followed to resolve names of records, tasks, protected objects, simple names and expanded names.

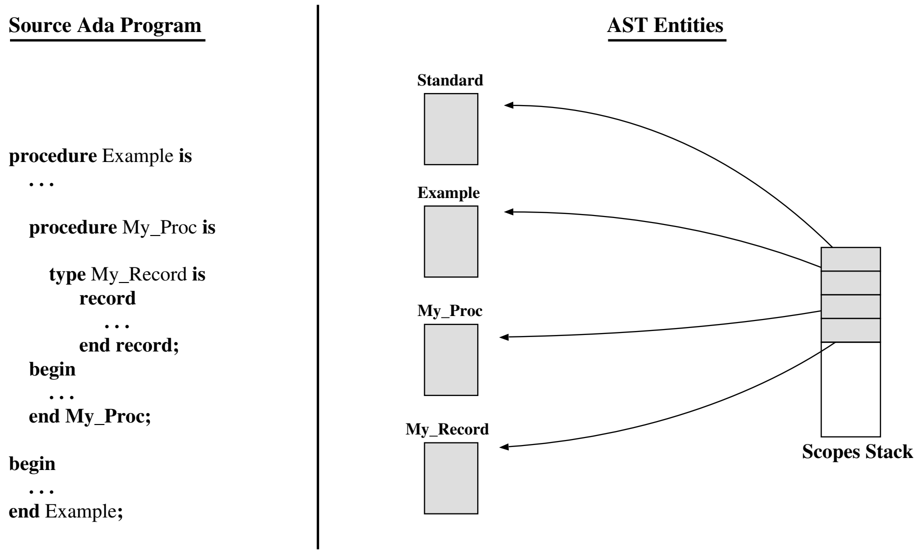
4.1 Flags and Data Structures

Entity nodes corresponding to defining entities have a set of flags and attributes (entity-node fields) which are used by the semantic analyzer to handle scopes and visibility. The basic flags used to verify the Ada rules of visibility are:

- **Is Immediately Visible**: The entity is defined in some currently open scope [AAR95, Section 8.3(4)].

- **Is Potentially Use Visible**: The entity is defined in a package named in a currently active use clause [AAR95, Section 8.4(8)]. Note that potentially use visible entities are not necessarily use visible [AAR95, Section 8.4(9-11)].

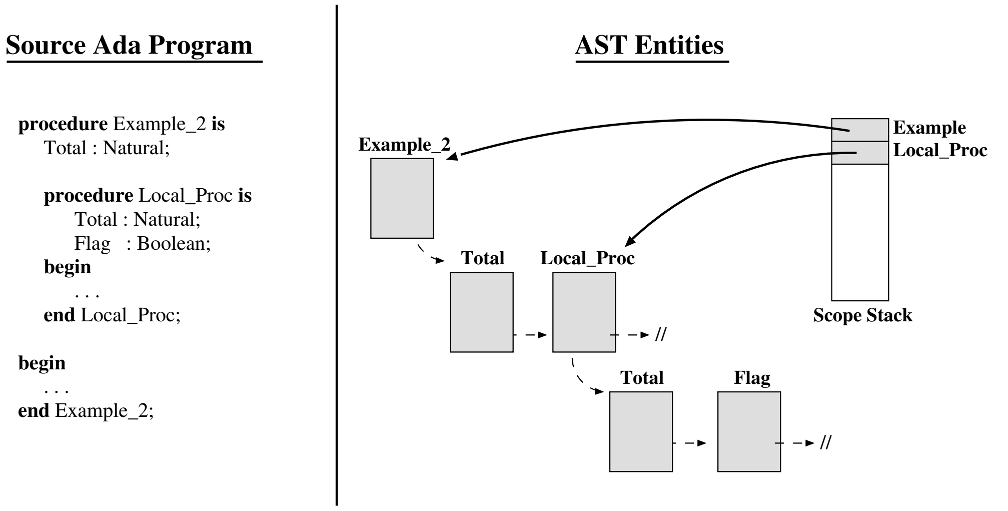
To keep track of the scopes currently been compiled the semantic analyzer uses one stack (*Scope_Stack*). Defining entities corresponding to declaration scopes (defining entities of packages, tasks, etc.) are placed in the scope stack while being processed, and removed at the end. The first element in the Scope Stack references the defining entity of package Standard. Figure 4.1 presents one example. On the left side we have a procedure (*Example*) which has a local procedure (*My_Proc*) with a local record type declaration (*My_Record*). The right side of this figure represents the contents of the Scope Stack at the point of analysis of the record type declaration. For simplicity, figures in the rest of this chapter do not have the reference to the standard package.



```
procedure Example is
   . . .
   procedure My_Proc is

      type My_Record is
          record
             . . .
          end record;
   begin
      . . .
   end My_Proc;

begin
   . . .
end Example;
```

Figure 4.1: The Scope Stack

The semantic analyzer links all defining entities declared in the same scope through the *Next_Entity* attribute. The entity which defines the scope has two attributes which reference the first and last entity in the scope. In addition, all entities have an attribute which references the defining entity of their scope. These attributes help the semantic analyzer to traverse entity scopes up and down. Figure 4.2 presents an example. On the left side we have an Ada program and, on the right side the corresponding data structure at the point of analysis of the sequence-of-statements of the local procedure (*Local_Proc*). For simplicity, only some semantic attributes are represented in the figures presented in this chapter.



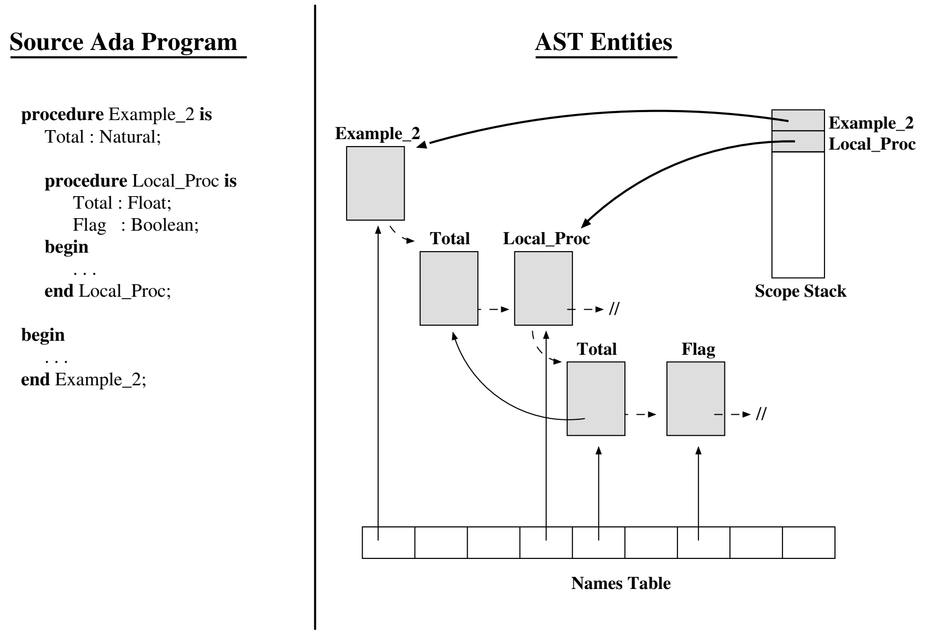
Source Ada Program

```
procedure Example_2 is
   Total : Natural;

   procedure Local_Proc is
      Total : Natural;
      Flag  : Boolean;
   begin
      . . .
   end Local_Proc;

begin
   . . .
end Example_2;
```

Figure 4.2: List of entities in the same scope.

Two entity definitions are *homonyms* if they have the same defining name and, if both are over-loadable, their profiles are type conformant. For example, an inner declaration hides any outer homonym declaration from direct visibility. The Ada Reference Manual uses the term *homograph* to refer to this concept [AAR95, Section 8.3(8)]; we have preferred to use the term homonym because it is the term used in the GNAT implementation. For example, the GNAT semantic analyzer keeps track of homonym entities by means of the *Homonym* linked lists. The head of each homonym list is saved in a general-purpose field of the *Names Table*. To provide fast access to names, the Names Table is a hash table with no duplicated names. In case of overloaded entities, the homonym list holds all the possible meanings of a given identifier. The process of overload resolution uses type information to select from this chain the unique meaning of a given identifier.

In the general case, the sequence of actions carried out by the Semantic Analyzer to handle scopes and visibility are: (1) Make immediately visible the defining entity of the new scope, (2) Open the scope, (3) Make immediately visible all the local entities defined inside the new scope, and (4) Insert all the local entities in the entity and homonym lists. Figure 4.3 presents one example. On the left side there is an Ada program with one global entity (*Total*) which has one homonym inside the local procedure *Local_Proc*. On the right side the reader can see the corresponding data structure after the local declarations have been analyzed and the entities have been inserted into the corresponding chains. In the case of *Total* we see the direct visibility of the local entity from the Names Table, and the link to its hidden homonym entity in the scope of *Example_2*).



```
procedure Example_2 is
   Total : Natural;

   procedure Local_Proc is
      Total : Float;
      Flag   : Boolean;
   begin
      . . .
   end Local_Proc;

begin
   . . .
end Example_2;
```

Figure 4.3: Lists of homonym entities.

When the analysis of the innermost scope is finished, the entities defined therein are no longer visible. If the scope is not a package declaration, these entities are never visible subsequently, and can be removed from visibility chains. If the scope is a package declaration, its visible declarations may still be accessible. Therefore the semantic analyzer leaves entities defined in such a scope in the visibility chains, and only modifies their visibility (immediate or potential-use visibility). This will be described in detail in Section 4.3.

In summary, all defining entities are chained twice: (1) In their scope, and (2) In their homonym list. As a consequence, the entities name space can be view as a sparse matrix where each row corresponds to a scope, and each column to a name. Open scopes, that is to say scopes currently being compiled, have their corresponding rows of entities in order, innermost scope first (cf. Figure 4.4).

In the rest of this chapter we will use the following expressions to refer to handling these flags and data structures: *"To make Immediately/Potentially-Use Visible"* an entity means to mark the entity as Immediately-Visible or Potentially-Use-Visible; the expression *"To open the scope"* of an entity will mean to push the reference of an entity in the Scope Stack, and finally the expression "*To insert*

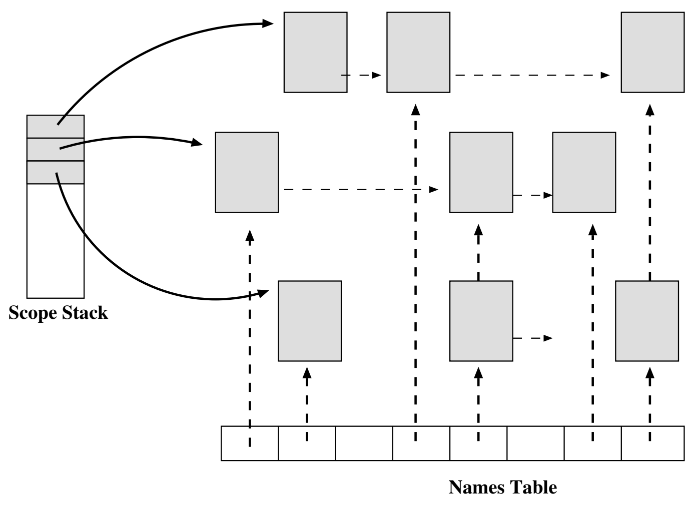


Figure 4.4: The matrix of entities.

an entity in the matrix of entities" means to chain the entity in the corresponding scope and homonym lists.

4.2 Analysis of Records, Tasks and Protected Types.

During the analysis of records, tasks, and protected types all their local entities must be immediately visible. Thus the semantic analyzer (1) makes immediately visible the defining entity of the record, task or protected type definition (2) opens the corresponding scope, (3) makes immediately visible all the entities defined inside the scope, and (4) insert them in the matrix of entities. The discriminants (if any) are handled as additional components of the type, and thus they are unchained on exit from their homonym lists. Figures 4.5 and 4.6 represent these steps during the analysis of a record type declaration and a protected type declaration. According to Ada, from the outside these local entities are always accessed as selected components (notation *Prefix.X*). Therefore, on exit from the scope the semantic analyzer (1) makes the defining entity not immediately visible, (2) closes the scope, and (3) removes the local entities from their homonym lists. As a consequence, further analysis of selected components requires to sequentially traverse the scope list named by *Prefix*. The analysis of the task-type and protected-type bodies will temporarily re-chain their local entities in the homonym lists.

```
procedure Example is
   Count : Integer;

   type My_Record is
      record
         Count   : Natural;
         Name    : String (1 .. 30);
      end record;
   . . .

begin
   . . .
end Example;
```

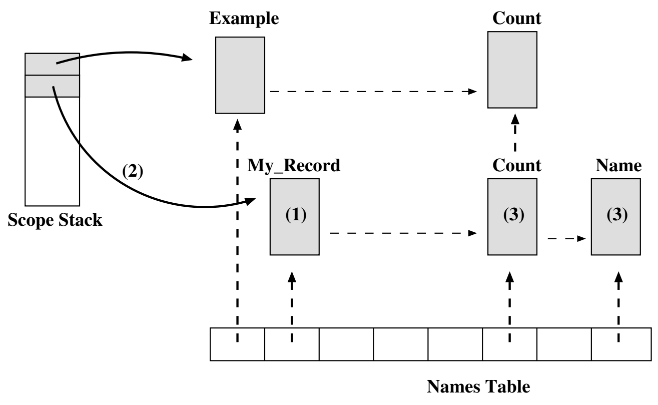


Figure 4.5: Analysis of a record.

4.3 Analysis of Packages

To analyze a package specification the semantic analyzer performs the following actions: (1) make immediately visible the defining entity of the package specification, (2) open the scope, (3) make immediately visible all the entities defined in the package specification, and (4) insert them in the matrix of entities. On exit the scope must be closed but, instead of removing all these entities from their homonym lists, the GNAT semantic analyzer just un-marks them as immediately visible. As a consequence, the analysis of the package body as well as subsequent with and use clauses do not need to re-chain the visible entities but only to re-open the scope and re-establish their direct visibility. Entities defined in the scope of the package body are just made immediately visible and added to the matrix of entities. Because the body is semantically an extension of the specification, these entities are added at the end of the list of entities of the package specification. To avoid visibility from the outside, on exit from the package body the entities defined in the package body are completely removed from the matrix of entities. Although these entities are not further required for analysis, they are saved in the in the defining entity of the package body because they are needed to generate the object code.

At a first sight, the analysis of *use_clauses* only needs to make Potentially-Use-Visible all the entities defined in the visible part of the named package specifications and, on exit from the scope of the use_clause, to reset the flag. However, this simple approach is not enough because it is legal to name the same package in nested contexts. To properly handle the general case, the defining entity of the

```
procedure Example is
  type T_Buffer is . . .
  Count : Natural := 0;

  protected type T_Stack is
    entry Push (X : in   Item);
    entry Pop   (X : out Item);
  private
    Count   : Natural := 0;
    Buffer   : T_Buffer;
  end T_Stack;

  protected body T_Stack is
    . . .
  end T_Stack;
  . . .
begin
  . . .
end Example;
```

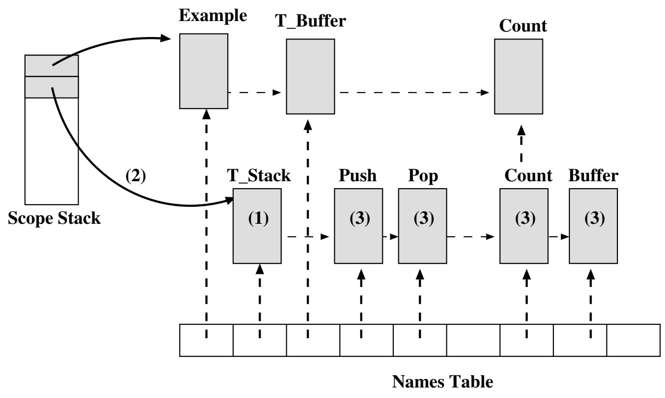


Figure 4.6: Analysis of a protected type.

package specification has a flag (*In_Use* which indicates it is currently in the scope of an use_clause. In case of redundant use, a second flag (*Redundant_Use*) is set. On exit from a scope *S*, the semantic analyzer scans in reverse order the list of use clauses in the declarative part of *S*. The visibility flag is clear as long as the package is not flagged as being in a redundant use clause, in which case the outer use clause is still in effect, and the direct visibility of its entities must be retained.

In the case of *use_type* clauses the Semantic Analyzer collects the primitive operators associated with the named types and makes them Potentially Use Visible.

4.4 Analysis of Private Types

Private types were Ada83's fundamental contribution to Software Engineering. Private types are the basic mechanism for encapsulation and information hiding: they separate the visible interface of a type (those properties that a client can use) from the implementation details of the type, which only the implementor of the type need to have access to. In one way or another, similar privacy mechanisms have found their way into other modern programming languages, in particular C++ and Java.

Ada95 has extended the privacy mechanisms to tasks and protected types. For protected types, information hiding means a complete separation between the operations that apply to the protected data, and the structure of this data. This is the modern realization of the notion of Monitor: the object provides some operations on completely private data. All the client needs to know is that there is a locking mechanism that ensures that the data is accessed in mutual exclusion.

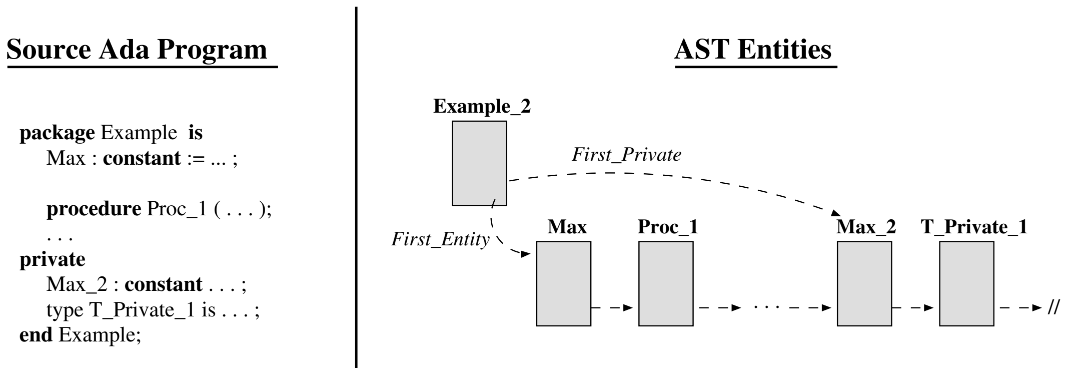
Ada95 has also extended the syntax of private type declarations to include private extensions and unknown discriminants. Finally, the management of privacy is central to the semantics of child units.

The salient characteristic of private types is that their description is given in two parts: a private type declaration introduces a partial view, which in general provides no details as to the structure of the type, except for a few characteristics that a client needs to know about (for example, whether the type has discriminants, or whether it is tagged). The full declaration provides all the structural information that the implementation of the type and its operations requires. In the compiler, the treatment of private types needs to cope with these two views. It is central to the organization of the compiler that a given entity is uniquely identified by a index into the nodes table. In other words, **the information concerning a given entity is always as the same location during a compilation**. To cope with the two-views description of private types, the compiler includes various mechanisms that exchange the partial and full views of these types depending on the context of the compilation. For example, when compiling a package body, the full view of all its types must be accessible, but when compiling a package specification that appears in a with_clause of some compilation unit, only the partial view of its private types must be visible to this unit. The view-exchange mechanism described below is conceptually straightforward, but its interaction with generics and inlining is the source of much complexity in the front-end.

The view-exchange mechanism applies to other entities that have two views, either because they have two declarations (such a deferred constants) or because their properties change with context (for example, a subtype of a private type becomes a subtype of the full view of the parent is visible, and a type with a private component may become non-private when the full view of the component type is visible).

4.4.1 Private Entities Visibility

To distinguish between visible and private package declarations, the defining entity of a package has an attribute that references the entity of the first private declaration of the package specification (cf. *First_Private_Entity* in Figure 4.7). This attribute is central to the implementation of the view-swapping mechanism, which is invoked at the end of a package body and at various points in the compilation of child units.


Source Ada Program

```
package Example  is
   Max : constant := ... ;

   procedure Proc_1 ( . . . );
   . . .
private
   Max_2 : constant . . . ;
   type T_Private_1 is . . . ;
end Example;
```

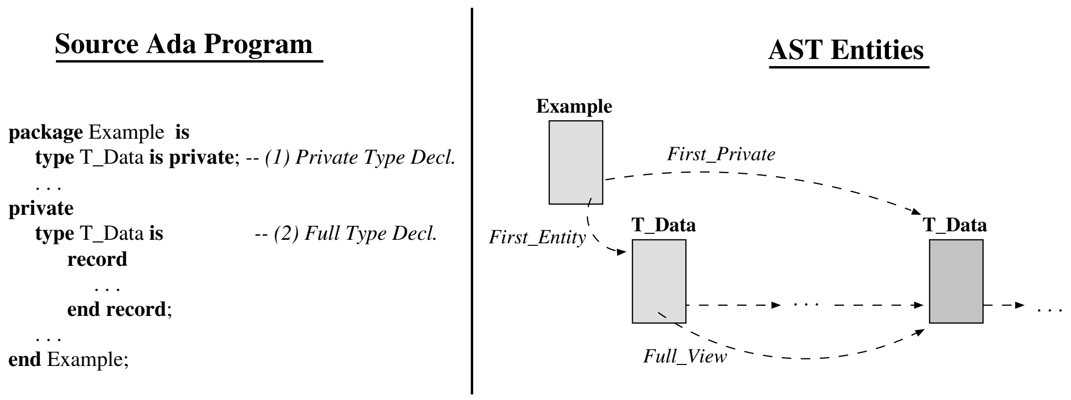
Figure 4.7: Visible and Private Entities

Similar to the package specification, the defining entity of the task type and the protected-object type also have the *First_Private_Entity* attribute to reference the first private entry (if any).

4.4.2 Private Type Declarations

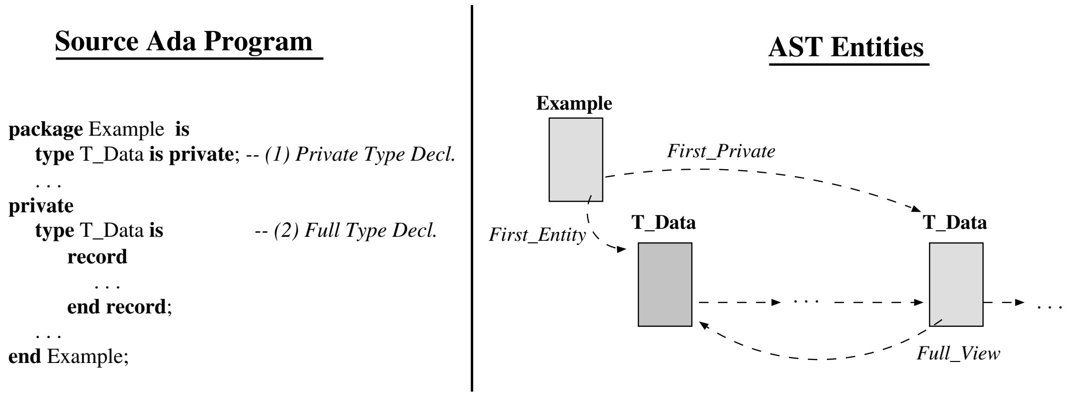
As mentioned above, the principle that each entity must have a single defining occurrence clashes with the presence of two separate declarations for private types, and thus (syntactically) two defining occurrences: (1) the private type declaration, which introduces the partial view, and (2) the full type declaration. The GNAT implementation treats the defining entity of the partial view as **the** entity for the type. This entity as an attribute *Full_View*, which denotes the entity of the full view. (cf. Figure 4.8). There is no link in the opposite direction, and all view-swapping activity starts from a partial view. Flag *Has_Private_View* is used to indicate that a given full type declaration is the completion of a private type declaration, and if need be corresponding partial declaration can be retrieved by a traversal of the public declarations of the package.

During semantic processing the defining occurrence also points to a list of private dependents, that is to say access types or composite types whose designated types or component types are subtypes or derived types of the private type in question. After the full declaration has been seen, the private dependents are updated to indicate that they have full definitions.



```
package Example  is
   type T_Data is private; -- (1) Private Type Decl.
   . . .
private
   type T_Data is              -- (2) Full Type Decl.
      record
         . . .
      end record;
   . . .
end Example;
```

Figure 4.8: Reference to the Full-View Entity



```
package Example  is
   type T_Data is private; -- (1) Private Type Decl.
   . . .
private
   type T_Data is              -- (2) Full Type Decl.
      record
         . . .
      end record;
   . . .
end Example;
```

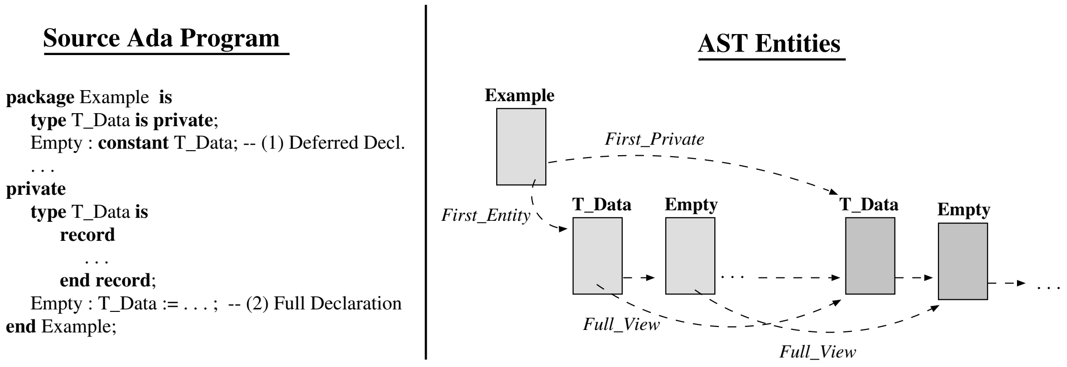
Figure 4.9: Swapping of the Private Declaration and the Full View

Ada95 allows task specifications to include a private part. Given that a task_item can only be an entry declaration or a representation_clause, this adds very little to semantic processing of tasks. When analyzing the task body, these declaration must be made visible. However, in this case there is no partial view, and therefore no view-swapping is needed.

The private part of a protected-type declaration is more significant from a semantic point of view: it encapsulates the state of an object of the type. When analyzing the protected body, the operations of the type are rewritten to include an implicit parameter, which is a record whose structure reflects the private part of the protected type. Thus, to each protected type we associate a record structure which is its *Corresponding_Record*. In turn, this record type has an attribute *Corresponding_Concurrent_Type*, that denotes the protected type from which it is defined. The Corresponding_Record eventually includes a run-time lock component that insures mutual exclusion. The processing of protected bodies is for the most part an expansion activity and is described in a separate chapter.

4.4.3 Deferred Constants and Incomplete Types

The mechanism described in the previous section is also used to handle deferred constant declarations and incomplete type declarations. Thus, the defining entity of a deferred constant declaration has a link to the corresponding full-type declaration entity (cf. Figure 4.10).



Source Ada Program

```
package Example is
   type T_Data is private;
   Empty : constant T_Data; -- (1) Deferred Decl.
   . . .
private
   type T_Data is
      record
         . . .
      end record;
   Empty : T_Data := . . . ; -- (2) Full Declaration
end Example;
```

Figure 4.10: Deferred Constants Handling

4.4.4 Limited Types

Limited types add no special complexity to the Semantic Analyzer; language-defined copying operations are just restricted for them.

4.4.5 Analysis of Child Units

The analysis of a child units requires to have special care with visibility of private entities. To analyze a public child package specification the Semantic Analyzer carries out the following actions: 1) make immediately visible all the entities the visible-part of the parent, 2) open the scope of the child package, 3) analyze the visible-part of the child package specification, 4) verify that incomplete types defined in the visible part have received their corresponding full declarations, 5) make immediately visible the parent private entities, and finally 6) analyze the private-part of the child package specification. In case of a private child package, the analysis is simpler because they have full visibility of the entities defined in the parent. Therefore, step 1 makes immediately visible all the entities declared in the visible and private parts of the parent, and follows steps 2, 3 and 5. The GNAT Semantic Analyzer has separate routines to make the visible and private declarations visible at different times (see *Install_Visible_Declarations* and *Install_Private_Declarations*).

4.4.6 Analysis of Subunits

Subunits must be compiled in the environment of the corresponding stub. That is to say with the same visibility and context available at the point of the stub declaration, but with the additional visibility provided by the context clauses of the subunit itself (if any). As a result, compilation of a subunit forces compilation of the parent. At the point of the stub declaration, the Semantic Analyzer is called recursively to analyze the proper body of the subunit, but without reinitializing the Names Table nor the Scope Stack (i.e. standard is not pushed on the stack). Thus, the context of the subunit is added to the context of the parent, and the subunit is compiled in the correct environment.

4.5 Name Resolution

Name resolution is the process that establishes a mapping between names and the defining entity referred to by the names at each point in the program. In the context of the GNAT semantic analyzer name resolution involves to link each node that denotes an entity with its corresponding defining-entity node. In case of simple names the semantic analyzer only uses the homonym list. If the entity is immediately visible, it is the one designated by the simple name. If only potentially-use-visible entities are found in the list, the semantic analyzer verifies they do not hide each other. In case of expanded names the semantic analyzer looks for the entity at the intersection of the entity list for the scope (the prefix) and the homonym list for the selector.

4.6 Summary

The GNAT front-end stores all the semantic information concerning program entities directly in defining entity nodes in the AST. Thus the AST contains not only the full syntactic representation of the program, but also the results of the semantic analysis. To handle the analysis of scope and visibility rules, all the entities have two flags which indicate if the entity is immediately or potentially-use visible. In addition, all the entities in the same scope are inserted in two lists: the list of entities in the scope, and the list of homonym entities. As a consequence, the entities name space can be view as a sparse matrix where each row corresponds to a scope, and each column to a name.

The semantic analyzer keeps all the entities defined in the package declaration in a list; visible entities are in the front of this list and private entities are in the rear. The semantic analyzer uses the reference to the first private entity as the limit of visible entities. This scheme is also used to implement the private entities in tasks and protected types. All the incomplete entities defined in the visible-part of the package specification (deferred constants, incomplete types and private types) have a reference to the corresponding full view in the private part. When the full-view is made visible, this link is used to swap the two declarations and thus make the full-definition available to the Ada programs.

Chapter 5

Overload Resolution

Ada supports the overloading of subprograms and operators [AAR95, Section 8.3]. This means that an occurrence of an identifier or operator (a *designator*) may denote several entities that are simultaneously visible at that point. Overloaded subprograms must differ in at least one of the following respects: (1) whether the subprogram is a procedure or function, (2) if the subprogram is a function, its result type, (3) the number of parameters, (4) the type of each parameter. These properties are collectively called the subprogram's *parameter-and-result-type profile*. For example, the following operations can be visible at the same point:

```
function  Op (x : Integer) return Integer;                 -- (Op1)
function  Op (x : Integer; y : Integer) return Integer;    -- (Op2)
function  Op (x : Float) return Integer;                   -- (Op3)
function  Op (x : Integer) return Float;                   -- (Op4)
procedure Op (x : Integer);                                -- (Op5)
```

Note that operations that only differ in the names of the formal parameters, but not their types, cannot be visible at the same point: either one hides the other through of scope and visibility rules, or else the declarations are illegal.

This notion is familiar from most other programming languages, where overloading is most commonly applied to operators. Overloading resolution is the process of selecting among the visible entities at a point (the *candidate interpretations* of the designator) the unique one that is compatible with the context in which it appears. In the simplest case, operators are resolved from the type of their operands: *(A+B)* denotes an integer addition or a floating-point addition, depending on the types of *A* and *B*. This simple rule applies to subprograms as well, where a candidate interpretation is retained if the types of the actuals in the call are compatible with the formals of the subprogram.

Ada, unlike many other programming languages, also uses the context of the call to resolve an overloaded name [AAR95, Section 8.6]. For example, a program may declare several functions whose signature differs only in their return type (cases (1) and (4) above). A consequence of the use of context is that type checking of expressions and the identification of operators cannot be performed in a single bottom-up traversal of the expression, as is the case for most other imperative languages. Instead, a two-pass algorithm is required (Algol68 had similar resolution rules, and also required a multi-pass resolution algorithm).

In the rest of this chapter we present the two-pass resolution algorithm implemented in GNAT. We first give a general description of the algorithm. Later we complete this description with additional details, and the data structures used to store the interpretations of an overloaded identifier.

5.1 Resolution Algorithm

The resolution algorithm has two main steps:

1. In a first step, we attach to each identifier the set of its candidate interpretations, and proceed to select from this set those interpretations that are compatible with the types of their arguments. If such a set has a single element, the designator is unambiguous.

   This upward pass is performed over each *complete context*, defined as a language construct over which overload resolution must be performed without examining a larger portion of the program. Each of the following constructs is a complete context: a context item, a declarative_item or declaration, a statement, a pragma_argument_association, and the expression of a case_statement [AAR95, Section 8.6(4-9)]. Procedure calls and assignment statements are thus examples of complete contexts.

2. In the second pass, type information is imposed on the complete context to select a single interpretation for each component of the construct. For example, the right-hand side of an assignment may be an overloaded function call, but if the type of the left-hand side is unique, it will identify uniquely the function being called. Once the function is known, the types of its formal parameters are used to resolve the possibly overloaded actuals in the call, and so on.

The problem of overload resolution is best explained by a simple example (a detailed description of this problem as well as a formal specification of the algorithm can be found in [vK87, Section 4.5]). Let us assume the following functions:

```
declare
   function F (A, B : T1) return T  is ...   -- (F1)
   function F (A, B : T2) return T1 is ...   -- (F2)
   function F (A, B : T2) return T2 is ...   -- (F3)
   Var1, Var2, Res : T2;
begin
   Res := F (Var1, Var2);
end;
```

The type of the expression *F(Var1,Var2)* cannot be deduced from the function call itself; the context in which it appears must be taken into account. In the first scan, the bottom-up analysis restricts at each node in the expression subtree the set of operators by discarding those operators whose formal-parameter types are inconsistent with the types of the actual-parameter expressions. In this example, the second and third version of *F* are valid candidate, so the processing attaches the set $\{F2,\ F3\}$ to the occurrence of *F* in the call. In the second pass, the top-down analysis restricts the set by discarding any operator or function whose result type is inconsistent with the context. In our example, the context requires the result type *T2*. Taking this into account, the top-down scan reduces the set of applicable functions to the *F* from the third definition.

5.1.1 Additional Details for the Bottom-Up Pass

The first pass can also select candidate interpretations on the basis of named parameter associations. Consider the two declarations:

```
function F (A: Integer; B: Integer := 0) return Integer;-- (F1)
function F (C: Integer) return Integer;                 -- (F2)
```

These two functions have different parameter profiles, but the call *F(5)* is ambiguous, regardless of the context: it can mean *F(5,0)* or *F(5)*. However, it is possible to write *F(C=>5)* to resolve the ambiguity because the named notation indicates that only *F2* is a candidate interpretation.

The details of overload resolution are complicated by construct-specific rules that apply to different complete contexts and also to constituents of a context, and finally by the existence of resolution rules that specify not a single type, but a class of types. For example:

1. The bounds of an integer type declaration can be of any integer type.
2. The condition in an if-statement can be of any boolean type.
3. The operands of an equality operator must be non-limited.

Note that in the course of overload resolution, expressions themselves can be seen as overloaded, in the sense that they have more than one candidate type. For example, a selected component can be overloaded if its prefix is overloaded.

The bottom-up pass labels the subtree as follows:

- Numeric literals are labelled with the corresponding universal type; character literals are labelled *Some_Character*, because they are compatible with any character type or enumeration type that has character literals.
- Aggregates, which are literal denotations for composite values, are labelled *Any_Composite*, indicating that the context must be a record or array type. Note that the components of an aggregate are not used in the bottom-up pass: the aggregate is resolved from the context.
- For identifiers, the visibility rules select an interpretation, and if the interpretation is overloadable (i.e. it is a subprogram or enumeration literal) then the full context is examined to locate other candidates interpretations. The homonym chain for the entity is traversed to locate entities with the same name that may be declared in outer scopes or in packages in the context of the current compilation. All of them are added to a data structure that is conceptually attached to the identifier, and which is described below (cf. Section 5.1.2).
- Each expression kind has specific resolution rules. In the semantic description of Ada, each type declaration creates some implicit operators that apply to the type. For example, every non-limited type has its own equality operator, each arithmetic type has its own arithmetic operators, etc. Given that type declarations are extremely common in Ada programs, it seems appropriate to find some compact or implicit description of these operators, rather than making explicit entities for all of them. This will be discussed in detail in the following paragraphs.

In GNAT, all predefined operators are declared only once, in the internal representation of package Standard. If the operands of an operator are overloaded, different interpretations correspond to different types but the name of the operator itself is unique. The absence of explicit operators for each type complicates somewhat type-checking, but represents a substantial saving in space and performance. Operators have the following characteristics:

1. The types of both operands must be the same.

2. The context does not determine the expected type of the operands, given that it only specifies that the result must be of type *Boolean*.

Consider the expression *f(x)=g(y)*, where both *f* and *g* are overloaded functions, and there are no user-defined equality operations, that is to say only the predefined equality is visible. Bottom-up analysis determines the possible interpretations of each call. We then look for types that appear in both sets of interpretations, that is to say we take the intersection of the candidate types of both operands. If this intersection contains more than one type, the construct is ambiguous regardless of context. If it contains a single type, the equality operation has a single interpretation with a boolean type.

In this section we have mentioned some typical resolution rules for expressions. Additional details can be found in the front-end packages *Sem_Ch4* and *Sem_Type*.

5.1.2 Data Structures

The semantic analyzer stores the interpretations of an identifier in a global overloads table. An interpretation consists of a pair (identifier, type). Expressions can also be overloaded, but do not have a name, so the corresponding interpretations are pairs that just repeat the type. Standard iterator primitives: *Get_First_Interp* and *Get_Next_Interp*, are used systematically in the analysis and resolution of overloaded constructs. A common idiom in type processing consists in checking whether a given overloaded node can be used in a given context. The routine *Has_Compatible_Type (T, N)* iterates over the interpretations of *N* and yields *True* if one of those has type *T*. Details can be found in the front-end package *Sem_Type*.

5.1.3 Additional Details for the Top-Down Pass

The second pass of type resolution traverses the AST from root to leaves, and propagates the type information imposed by the context to each subcomponent of the context. For example, given the following:

```
declare
   procedure P (X : Float; Y : Integer);              -- P1
   procedure P (X : Float; Y : Float);                -- P2
   function  F (X : Float) return Integer;            -- F1
   function  F (X : Float) return Float;              -- F2
   ...
begin
   P (F (5.0), 2.2);
end;
```

The bottom-up pass determines that the call to *F* has the set of possible interpretations: $\{F1,F2\}$. Similarly, the identifier *P* is either *P1* or *P2*. The analysis of the call itself selects *P2*, which in turns determines that the type of the first actual must be *Float*. This type information is then applied to the call *F(5.0)* to select *F2*.

Each language construct has a corresponding subprogram in the front-end package *Sem_Res*, that applies the context information to the constituents of the construct. For example, consider an overloaded indexed expression:

F(G(x))(H(y))

...where *F*, *G* and *H* are overloaded function calls. Once the context type *CT* is known, we iterate over the interpretations of *F* and retain the one that returns an array type whose component type is *CT*. Once a unique interpretation of *F* is known, its formal parameter provides the unique type to resolve *G(x)*, and the component type of its return type is used to resolve *H(y)*.

The language displays a syntactic ambiguity which requires special processing. Consider the following declarations:

```
type Vector is array (Integer range <>) of Integer;
function F (X : Integer := 10) return Vector;
```

The expression *F(5)* has two possible interpretations: a function call with parameter *5* that returns a vector, i.e. *F(X=¿5)*, or the indexing of a parameterless function call that uses the default value, i.e. *F(x=¿10)(5)*. These two interpretations have different AST's, and the proper interpretation is obtained from context: the expected type of the call is either a *Vector* or an *Integer*. However, it is awkward to carry two different trees for this construct. Rather, the parser builds the first interpretation, and the resolution of function calls looks for this particular case (see details in *Analyze_Call* and *Resolve_Call*.

5.2 Summary

Resolution of overloaded subprograms and operators requires a two-pass algorithm. The first pass attaches to each identifier the set of its candidate interpretations with interpretations are compatible with the types of their arguments. If such a set has a single element, the designator is unambiguous. In the second pass, type information is imposed on the complete context to select a single interpretation for each component of the construct. This analysis is performed over each *complete context*, defined in Ada as a language construct over which overload resolution must be performed without examining a larger portion of the program.

Chapter 6

Analysis of Discriminants

A discriminant is a special component that is used to parametrize objects of a composite type. The components of a discriminated type can depend on the value of the discriminants of the object: for example the constraints on the subtype of a component, or the initial value of a component, can be given by the value of a discriminant of the object. Discriminants are also used to describe variants of a variant record type, that is to say the determine the existence of other record components. With the exception of array types, all composite types in Ada can have discriminants. Thus, record types, protected types and task types, as well as the corresponding subtypes, may have discriminants.

A type with discriminants is unconstrained, that is to say, it does not have sufficient information to build an object of the type; a declaration of an object of such a type must supply values for the discriminants, by means of a discriminant constraint. For example:

```
type My_Record (Max_Length : Positive) is
record
   Name : String (1 .. Max_Length);
end record;

Obj : My_Record (30);
```

Because of the properties of components depend on the values of the discriminants, a discriminated object is self-consistent: the value of a discriminant implies the truth of some invariant for the object, for example the size of an array component. For that reason the values of discriminants cannot be changed arbitrarily, independently of the values of the components that depend on them. As a result the discriminants of an object must be treated semantically as constants, and unlike other record components they cannot be modified by a component assignment, or passed as out parameters in a call.

The above makes it clear that discriminated types are a parametrization mechanism, and that the discriminants are handled like parameters when creating a discriminated object. This parallel extends to the syntax and semantics of discriminant specifications and formal parameter specifications.

Like subprogram parameters, discriminant specifications may include a default expression. If the discriminants of a record have defaults, it is possible to declare an object of the type without providing an explicit constraint, in which case the object takes its discriminant values from the corresponding default expressions. Such an object is said to be unconstrained, and it is possible to modify the object by means of an assignment to the object as a whole, that modifies the discriminant values as well as those of the components that depend on them.

Discriminants serve similar purposes for tasks and protected types. In both cases, they can be used to constrain components as well as entry families. It is also common to use a discriminant to specify the priority of a task object, so that different objects of the same task type have different priorities.

In Ada83 discriminants must be of a discrete type. This reflects the common use of a discriminant as the expression in a case statement that describes the variants of a given record type. Ada95 introduces the notion of an access discriminant, which allows an object to be parametrized by a pointer to another object. Such access discriminants cannot be used to govern a variant, because they are not discrete. A type with an access discriminant is a limited type, because assignment is not meaningful for objects that contain pointers to other objects.

6.1 Analysis of Discriminants

At a first sight the analysis of discriminants adds no special complexity to the compiler. The immediate scope of the discriminants is the type definition, which includes the declarations of the remaining record components, but excludes the discriminant specification itself. It would appear that the Semantic Analyzer just needs to (1) enter the discriminants into the scope of the type declaration, (2) verify that default expressions are provided either for all or for none of the discriminants [AAR95, Section 3.7(11)], and (3) verify that the discriminant names are not used in the default-expressions of other discriminants. However, things are invariably more complicated.

For example, default expressions must be analyzed in a special fashion, because they correspond to *per-object constraints* [AAR95, Section 3.3.1(9)]: each object that is declared without explicit discriminants must evaluate the defaults anew. That is to say, if the default is a function call, the call must be executed for each object of the type, and not just when the type declaration is analyzed. However, the compiler must perform all legality checks at the point of type definition. As a result, analogous to the way in which generic units are analyzed, default expressions in the type declaration are analyzed in a special mode that excludes expansion. This analysis leaves the default expression marked as unanalyzed, although the semantic analyzer evaluates static expressions and performs related freezing operations (cf. detailed comment in package *Sem*). The semantic analyzer attaches this pre-analyzed expression to the defining entity of the discriminant, so it can easily retrieve it at points of use, that is to say when unconstrained objects of the type are declared.

Another aspect of semantic analysis that is complicated by the presence of discriminants is the handling of aggregates [AAR95, Section 4.3]. An *aggregate* for a record type must provide values for all components of the type. Therefore, if the type has discriminants, their values must be supplied as well. If the type has a variant part, the aggregate must specify all the components of the particular variant that corresponds to the given values of the discriminants. To allow the compiler to identify the specified variant part and gather the components that must be present, the discriminants in such an aggregate must be static. In case of nested variant parts the semantic analyzer must recursively traverse the record type structure to verify that the expressions corresponding to each discriminant ruling the nested variant parts are static, see what variants are selected by the given discriminant values, and verify that a value is given for all the components in that variant. For example, let us consider the following example:

```
 1: declare
 2:    type T_Company is (Small, Big);
 3:    type T_Record (Company_Kind     : T_Company;
 4:                   Num_Departments  : Natural) is
 5:    record
 6:       Num_Workers : Positive;
 7:       case Company is
 8:           when Small =>
 9:                Has_Benefits : Boolean;
10:           when Big =>
11:                case Num_Departments is
12:                    when 1 .. 10 =>
13:                        Value : Integer;
14:                    when others =>
15:                         null;
16:                end case;
17:       end case;
18:    end record;
19:
20:    Obj_1 : T_Record (Small, 1);
21:    Obj_2 : T_Record (Big, 7);
22:    Obj_3 : T_Record (Big, 15);
23: begin
24:    Obj_1 := (Num_Workers => 2,   Has_Benefits => False);
25:    Obj_2 := (Num_Workers => 100, Value => 10000);
26:    Obj_3 := (Num_Workers => 150, Value => 15000); -- ERROR
27: end;
```

The example presents a record-type declaration with nested variant parts (lines 7 to 17), three object declarations (lines 20 to 22), and three statements which initialize the objects by means of aggregates (lines 24 to 26). The third aggregate is wrong because, at the point of the object declaration (line 22) the number of departments was contrained to 15, and this value is used in the nested variant part (lines 11 to 16) to specify that no additional information is required for this kind of company.

To handle the general case, the analysis of a record aggregate proceeds by building a map that associates each component with the corresponding expression in the aggregate. The map contains at first the discriminants themselves, and eventually all components that appear in the selected variants (see details in package *Sem_Aggr*, and in subprogram *Gather_Components*).

Extension aggregates for type extensions add further complexity to the analysis, because the components may come from ancestors of the given type. In this case the values of discriminants are used to traverse variant parts of the ancestors to collect the list of required components. In all cases the analyzer verifies that values have been provided for all components, that they have the proper types, and that no extraneous values are present.

6.2 Analysis of Discriminants in Derived Types

When a type with discriminants is derived, the discriminants of the parent type can be inherited, constrained in the derivation, or renamed by the introduction of new discriminants. The following are the basic rules for derivation [AAR95, Sections 3.4(11), 3.7(13-15)]:

- If no discriminant is specified, the derived type inherits the discriminants of the parent type (implicitly in the same order and with the same specifications and defaults). In this case the derivation simply copies the full declaration of the parent, and the components of the derived type are in one-to-one correspondence with those of the parent.
- If the derived type has discriminants, and the parent is not a tagged type, each discriminant of the derived type shall be used in the constraint defining a parent subtype.

The last rule guarantees that in the absence of representation clauses, the layout of an untagged derived type is identical to that of its parent. The code generator only needs to refer to the physical layout of the original type. To handle the renamings that may be introduced by the derivation, the defining entity of the discriminant in the derived type includes an attribute *Corresponding_Discriminant* that allows the Semantic Analyzer to find the original discriminant in the parent type (cf. Comment in *Build_Derived_Record_Type* implementation).

In the presence of representation clauses, this simple model is not viable. because Ada allows the use of representation clauses in a derived untagged type *D* that specify a different record layout from that its *parent* type *P*. Hence the corresponding component can be placed in two different positions in the parent and in the derived type. As a result, the two types cannot share the declarations of their components, but must have fully disjoint complete declaration trees. The GNAT semantic analyzer does a copy of the entire tree for component declarations of *P* and builds a full type declaration for derived type *D*. Hence *D* appears as a record type of its own, with its own representation clauses, if any. The entity for *D* indicates that this is a derived type, and points to the parent subtype *P*.

Representation clauses cannot be provided for tagged types because dispatching and polymorphism mandate the same representation for the common components of the entire class.

6.3 Discriminals

The analogy between discriminants and parameters is even more apparent when we examine object initialization. If a composite type has components that have an implicit or explicit initial expression, objects of the type must be initialized at the point of creation. For this purpose, the compiler generates initialization procedures for each such type, and invokes this procedure each time an object of the type is declared. If the type has discriminants, the initialization procedure has parameters that are in one-to-one correspondence with the discriminants. Within GNAT, these parameters are called **discriminals**, and there is a semantic link between a discriminant and its corresponding discriminal. The call to the initialization procedure includes a parameter list which is a copy of the discriminant values used (implicitly or explicitly) in the object declaration. Details of initialization procedures are discussed in a separate chapter. For example, let us consider the following type declaration:

```
type Rec (D : Integer) is record
   Value : Integer := D;
   Name  : String (1 .. D) = (1 .. D => '!');
end record;
```

The corresponding initialization procedure is as follows:

```
procedure Init_Proc (Obj : in out rec; D : Integer) is
begin
   Obj.Value := D;
   Obj.Name  := (1 .. D => '!');
end;
```

The declaration *Obj1 : rec (17)* results in the generation of the call:

Init_Proc (Obj1, 17).

6.4 Summary

A discriminant is a special component that acts as a parameter for objects of the type. A default expression of a discriminant has a special significance: it allows objects of the type to be unconstrained. To properly handle discriminants the Ada compiler must takes special care with the analysis of default expressions, which must be evaluated at the point of the object declaration, not at the point of the type declaration. In addition, because the Ada representation clauses allow a derived type to specify a completely different record layout from its parent type, the derived type must copy all the components of the parent type. The analysis of record aggregates must use the specified values for the discriminants to determine the existence and properties of the remaining components of the object, in order to verify the semantic correctness of the aggregate as a whole.

Chapter 7

Generic Units

A generic unit is a parameterized template for a subprogram or package whose parameters can be types, variables, subprograms and packages. Generic units must be instantiated, by suppling specific actuals for the parameters. Each instantiation conceptually creates a copy of the specification and body of the generic unit, which are appropriately customized by the actual parameters.

The semantics of Ada specify that the legality of generic templates is established at the point of declaration of the generic, and not at the point of instantiation. Therefore, a generic unit must be fully analyzed at the place it appears. For an instantiation, the main legality checks involve verifying that the actuals provided in the instance match the generic formal parameters. The so-called "contract model" [AAR95, Section 12.3.1(a)] states that if the generic declaration is legal, and the actuals match the formals, then the instantiation is known to be legal, and semantic checks need not be performed on it.

The legality checks on a generic unit are similar to those of a regular, non-generic unit. They mainly involve name resolution and type checking. Given that the semantics of the generic are established at the point of definition, name resolution involves global name capture: a reference to an entity that is external to the generic is established when the generic is analyzed, and must be retained for each instantiation. On the other hand, entities local to the generic will have a different meaning in each instance.

Traditionally, there are two implementation techniques for generic instantiation [SB94]: in-line expansion (often called *Macro Expansion*) and direct compilation of generic units (also known as *Shared Generics*). The latter model is known to be fraught with difficulties and GNAT, like most other compilers, implements generics by expansion. Therefore, each instantiation of a given generic produces a copy of the generic code, suitably decorated with semantic information from captured global entities and actual parameters. Note that this is quite distinct from the macro-expansion of older programming languages, which is typically a pure textual substitution with no semantic legality checks on the macro itself.

After analysis, a generic unit contains partial semantic information. At the point of instantiation, the semantic information must be completed. This does require a full pass of semantic analysis over the instance. In addition, code expansion must be performed. Note that expansion cannot be performed on the generic itself, because expansion typically depends on type information that is not available until the actual types are known. Given that semantic analysis and expansion are inextricably linked in GNAT, we must perform full semantic analysis of the instance, even though the contract model would indicate that some of it is superfluous.

The requirements of global name capture determine the architecture described in this chapter. This architecture is complicated by two related aspects of Ada: nested generic units (and instantiations within generic units) and generic child units. In addition, private types complicate visibility analysis and require some special machinery to insure that name resolution is consistent between the point of generic definition and the points of instantiation.

This chapter is structured as follows: Section 7.1 presents the main details of the GNAT approach to the analysis and instantiation of simple generic units. This section also describes parameter matching and private types handling. Section 7.2 discusses the analysis and instantiation of nested generic units; Section 7.3 presents the analysis and instantiation of generic child units; Section 7.4 describes the instantiation of bodies in a separate phase, and Section 7.5 briefly discusses the mechanisms used to detect instantiation circularities. We close with a brief summary of the contents of this chapter.

7.1 Generic Units

7.1.1 Analysis of Generic Units

Generic specifications are analyzed wherever they appear. If a generic unit is mentioned in a context clause of some unit U, it is analyzed before the unit U itself. The result of this analysis is an Abstract Syntax Tree (AST) annotated with partial semantic information. The analysis of a generic package or subprogram specification involves the following sequence of actions:

1. **Copy the AST of the generic unit**. The first step is to create a copy of the unanalyzed AST of the generic. The generic copy is a tree that is isomorphic to the original. Semantic analysis will be performed on the copy, and only the relevant information concerning references to global entities will be propagated back to the original tree. In order to perform this propagation, each AST node in the original that may eventually contain an entity reference (typically identifiers and expanded names) holds a pointer to the corresponding node in the copy. The left side of Figure 7.1 represents the AST associated with an Ada program: an Ada compilation unit that has an inner generic unit (indicated by the dotted rectangle). Circles represent non-entity AST nodes (expressions, control structures, etc); rectangles represent AST entities. Dark circles or rectangles represent nodes with semantic information (decorated nodes). The left side of the figure represents the AST of the current compilation unit.

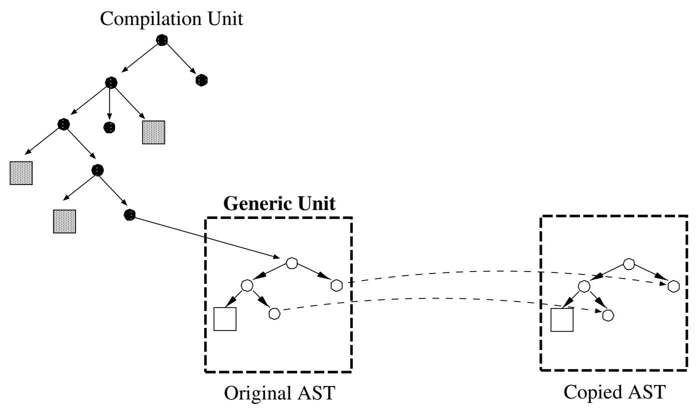


Figure 7.1: Step 1: Copy the original generic AST.

2. **Analyze the copy**. The copy is semantically analyzed to perform name resolution and all the required legality checks. Thus the copied AST is annotated with full semantic information. Figure 7.2 shows the decorated copy. The two arrows in the decorated AST (right side of the figure) represent a reference to a global entity, declared outside the generic unit, and a reference to one local entity, declared in the scope of the generic unit itself.

3. **Save non-local references**. After semantic analysis, the front-end traverses the original AST to identify all references to non-local entities (entities declared outside the generic itself). This is done by examining the scope of

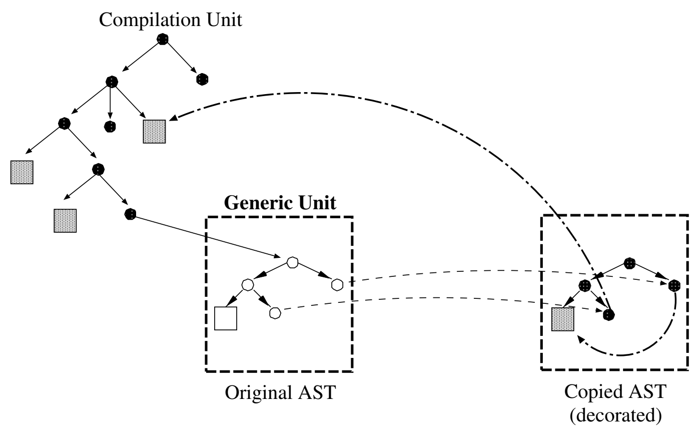


Figure 7.2: Step 2: Analyze and decorate the copy.

definition of all entity references, and determining whether they occur outside of the current generic unit. References to local entities are simply removed from the original AST. In this fashion, only global entity references will be propagated from the original AST to the instances. In the left side of Figure 7.3 the original node corresponding to a global entity reference is now represented in black; this means it is semantically decorated and thus its reference to the global entity will be preserved in all instances of this generic unit.

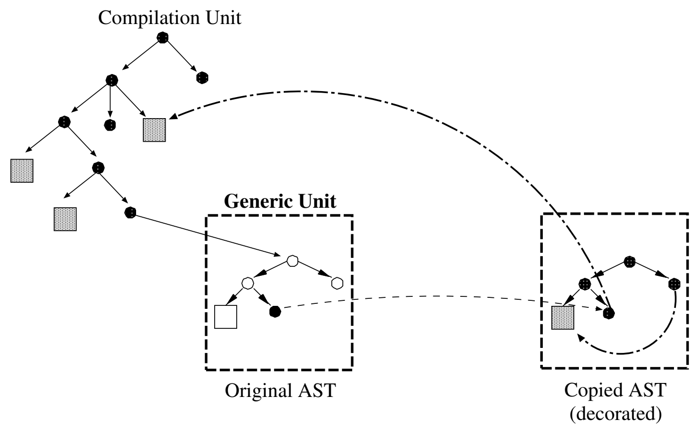


Figure 7.3: Step 3: Remove local references in the original AST.

7.1.2 Instantiation of Generic Units

At the point of generic instantiation, the original generic tree for the generic unit is copied, and the relevant semantic information from the copy is retrieved and added to it (cf. Figure 7.4). The new tree is treated as a regular (non-generic) unit which is analyzed and for which code is now generated. Non-local references are not analyzed anew (name resolution does not modify an entity reference that already denotes an entity, see *Find_Direct_Name*). To each local entity in the generic there corresponds a local entity in each instance, and name resolution for these is performed as for non-generic units.

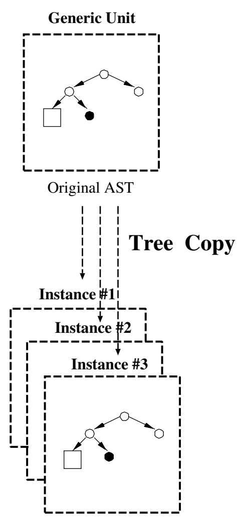


Figure 7.4: Instantiation of generic units.

7.1.3 Parameter Matching

The conventional model of instantiation is to replace each reference to a formal parameter by a reference to the matching actual. This is usually carried out by establishing a mapping from one set to the other, and applying this mapping to a copy of the generic template. This mapping approach is complex and expensive when nested generics are present, and GNAT follows a different approach, which closely reflects the semantics of instantiation. To indicate that a reference to a formal type is really a reference to the actual, the front-end inserts renaming declarations within the instance so that any use of the name of a formal resolves to the corresponding actual. The nature of the renaming declaration depends on the generic parameter: object, type, formal subprogram or formal package. Thanks to this mechanism, the analysis of the instance can proceed as if the copy had been inserted literally at the point of instantiation, even though the visibility environment at this point bears no relation to the visibility at the point of generic declaration.

1. **Objects**. For each generic *in* parameter the GNAT front-end generates a constant declaration; for each generic *in out* parameter generates an object renaming declaration whose renamed object is the corresponding actual (*out* mode formal objects are not allowed in Ada 95 [AAR95, Section 12.4-1a].
2. **Types**. Generic types are renamed by means of subtype declarations.
3. **Formal subprograms and packages**. For each formal subprogram or formal package the front-end generates the corresponding renaming declaration.

To give the right visibility to the entities introduced by these renamings, GNAT uses a different scheme for package and subprogram instantiations:

- **Packages**. In the case of packages, the list of renamings is inserted into the package specification, ahead of the visible declarations of the package. The renamings are thus analyzed before any of the text of the instance, and are visible at the right place. Furthermore, outside of the instance, the generic parameters are visible and denote their corresponding actuals. For example:

```
generic
   In_Var    : in Integer;
   InOut_Var : in out Integer;
   type T_Data is private;
package Example is
   type T_Local is ...
   procedure Dummy;
end Example;
```

```
package body Example is
   procedure Dummy is
      Aux : Example.T_Local;
   begin
      null;
   end Dummy;
end Example;
```

```
with Example;
procedure Use_Example is
   My_Var_1 : Integer := 1;
   My_Var_2 : Integer := 2;

   package My_Instance is new Example
      (In_Var     => My_Var_1,
       InOut_Var => My_Var_2,
       T_Data     => Integer);
begin
   null;
end Use_Example;
```

The GNAT front-end generates the AST equivalent of the following code for the instantiation:

```
procedure use_example is ------Front-end Translation

  my_var_1 : integer := ... ;
  my_var_2 : integer := ... ;

  package my_instance is
    in_var     : constant integer := my_var_1; -- (1)
    inout_var : integer renames my_var_2;     -- (2)
    subtype t_data is integer;                -- (3)
    type T_Local is ...

    package example renames my_instance;       -- (4)
    procedure dummy;
  end my_instance;

  package body my_instance is
    procedure dummy is
      Aux : Example.T_Local;                   -- (5)
    begin
      null;
    end dummy;
  end my_instance;

begin
   null;
end use_example;
```

Line (1) is the constant declaration corresponding to the **in** parameter; line (2) is the renaming of the **in out** parameter; line (3) is the subtype declaration used to rename the generic formal type *T_Data*. Finally, line (4) introduces a renaming for the generic unit itself. This corresponds to the fact that within the generic, the name of the generic denotes the name of the current instance. The need for this last renaming is made evident by the declaration in line (5), which references a type that is originally local to the generic, and now designates a type local to the current instance.

- **Subprograms**. In this case there is no obvious scope in which the required renaming declarations can be inserted. Instead, the front-end creates a wrapper package that holds the renamings followed by the subprogram instance itself. The analysis of this package makes the renaming declarations visible to the subprogram. After analyzing the package, the front-end inserts a subprogram renaming outside the wrapper package to give it the proper visibility (thus the callable subprogram instance appears declared outside the wrapper package). For example:

```
generic
   In_Var    : in Integer;
   InOut_Var : in out Integer;
   type T_Data is private;
function Example (A: T_Data) return Integer;
```

```
function Example (A: T_Data) return Integer is
begin
   return 1;
end Example;
```

```
with Example;
procedure Use_Example is
   My_Var_1 : Integer := ... ;
   My_Var_2 : Integer := ... ;

   function My_Instance is
     new Example (In_Var    => My_Var_1,
                  InOut_Var => My_Var_2,
                  T_Data    => Integer);

begin
   null;
end Use_Example;
```

In this case the GNAT front-end carries out the following transformation:

```
procedure use_example is  ------Front-end Translation
   my_var_1 : integer := ... ;
   my_var_2 : integer := ... ;

   package my_instanceGP110 is
      in_var     : constant integer := my_var_1;  -- (1)
      inout_var : integer renames my_var_2;       -- (2)
      subtype t_data is integer;                  -- (3)

      function my_instance (a : t_data) return integer;
   end my_instanceGP110;

   package body my_instanceGP110 is
      function example (a : t_data)
         return integer renames my_instance;      -- (4)

      function my_instance (a : t_data) return integer is
      begin
         return 1;
      end my_instance;
   end my_instanceGP110;

   function my_instance                           -- (5)
      (a : my_exampleGP110.t_data)
      renames my_instanceGP110.my_instance;
begin
   null;
end use_example;
```

Line (5) is the subprogram renaming which makes the subprogram visible in the current scope. The reader should note that this renaming declaration can not be replaced by an Ada *use_clause* on the wrapper package, because this would render visible all the renamings associated with formal parameters (not only the instantiated subprogram), which would be illegal.

If the subprogram instantiation is a compilation unit, the front-end gives the wrapper package the name of the subprogram instance. This ensures that the elaboration procedure called by the GNAT *Binder*, using the compilation unit name, calls in fact the elaboration procedure for the package.

7.1.4 Private Types

The analysis of a generic unit leaves all non-local references decorated to ensure that the proper entity is referenced to in the instantiations (cf. Section 7.1.1). However, for private types this by itself does not insure that the proper *View* of the entity is used (the full type may be visible at the point of generic definition, but not at instantiation time, or vice-versa). To solve this problem the semantic analyzer saves both views. At time of instantiation, the front-end checks which view is required, and exchanges declarations, if necessary, to restore the correct visibility. (cf. Chapter 4). After completing the instantiation, the front-end restores the current visibility.

7.2 Nested Generic Units

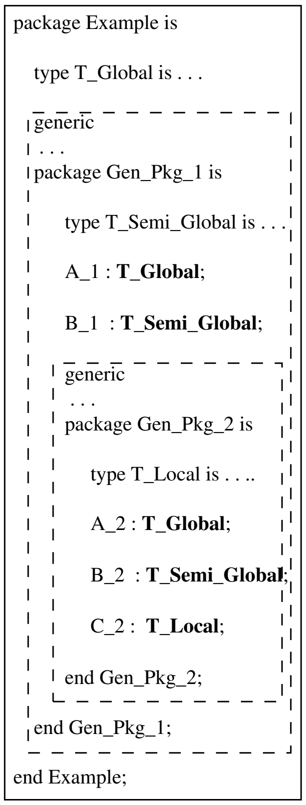
7.2.1 Analysis of Nested Generic Units

The model presented in the previous section extends naturally to nested generics. If a generic unit *G2* is nested within *G1*, the analysis of *G1* generates a copy *CG1* of the tree for *G1*. Later, when *G2* is analyzed, the front-end makes a copy *CG2* of the tree for *G2*, and establishes links between this copy and the original tree for *G2* within *CG1*. An instantiation of *G2* within *G1* will use the information collected in *CG2*. To see this in detail, consider the scheme in Figure 7.5 (Ada elements which reference entities in the AST have been emphasized in Figure 7.5 with **bold** letters).

This example shows a non-generic package (*Example*) with two nested generic packages (*Gen_Pkg_1* and *Gen_Pkg_2*). Data type *T_Global* is external (global) to the generic packages; data type *T_Semi_Global* is local to *Gen_Pkg_1* but external to *Gen_Pkg_2*. In addition, some objects of these data types are created in each generic package (*A_1, B_1, A_2, B_2* and *C_2*).

To do the semantic analysis of *Gen_Pkg_1* the GNAT front-end makes a copy of its AST and links the nodes which reference entities with their copy (cf. Figure 7.6). The semantic analysis of *Gen_Pkg_1* captures all occurrences of local and non-local entities (see the arrows from *A_1* to *T_Global*, and from *B_1* to *T_Semi_Global*).

To analyze *Gen_Pkg_2* the front-end repeats the process: makes a new copy of *Gent_Pkg_2* AST and links the nodes which reference entities with their new copy



```
package Example is

   type T_Global is . . .

   generic
   . . .
   package Gen_Pkg_1 is

      type T_Semi_Global is . . .

      A_1 : T_Global;

      B_1  : T_Semi_Global;

      generic
       . . .
      package Gen_Pkg_2 is

         type T_Local is . . ..

         A_2 : T_Global;

         B_2  : T_Semi_Global;

         C_2 :  T_Local;

      end Gen_Pkg_2;

   end Gen_Pkg_1;

end Example;
```

Figure 7.5: Nested generic packages.

(cf. Figure 7.7). Again, its semantic analysis captures all occurrences of local and non-local entities.

After the analysis of the nested generic units the front-end removes all the local references in the original copy of the nested generic units (cf. Left side in Figure 7.8). References to *T_Semi_Global* are not preserved because, in the general case, they may depend on generic formal parameters of *G1*.

7.2.2 Instantiation of Nested Generic Units

If a generic unit *G2* is nested within *G1*, the Ada programmer must first instantiate *G1* (say *IG1*) and then instantiate the nested generic unit *G2* which is inside *IG1*.The instantiation of *G1* produces a full copy of its AST, including the inner generic *G2*.

Continuing with our previous example, let us consider the following instances

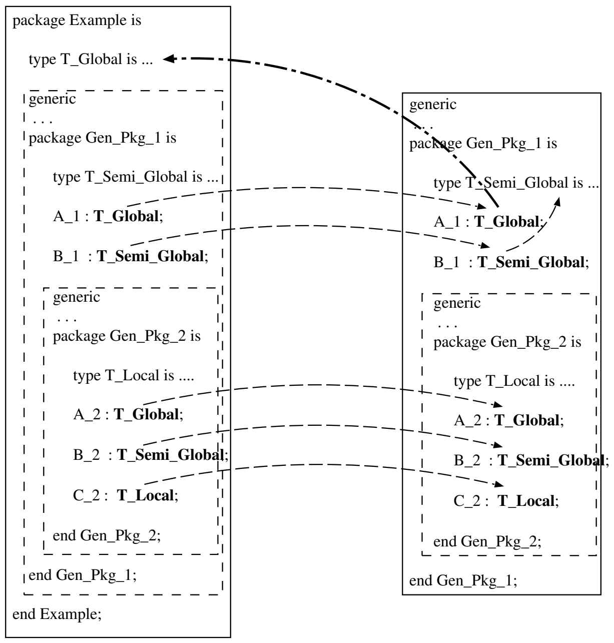


Figure 7.6: Analysis of generic package *Gen_Pkg_1*.

of our generic packages *Gen_Pkg_1* and *Gen_Pkg_2*:

```
with Example;
procedure Instances is
  package My_Pkg_1 is
     new Example.Gen_Pkg_1 (...);                          -- 1
  package My_Pkg_2 is
     new My_Pkg_1.Gen_Pkg_2 (...);                         -- 2
begin
  ...
end Instances;
```

At point 1 (instantiation of *Gen_Pkg_1*), the analysis of this instantiation includes the analysis of the nested generic, which produces a generic copy of *Gen_Pkg_2*, which will include global references to entities declared in the instance. The instantiation at point 2 requires no special treatment (the front-end produces a copy of the generic unit *Gen_Pkg_2* in the instance *My_Pkg_1* and instantiates it as usual).

Figure 7.9 presents the resulting code for this example. The Ada source lines which instantiate the generic units have been kept in the figure as Ada comments to help the reader see the corresponding instantiation (inside dotted rectangles). The

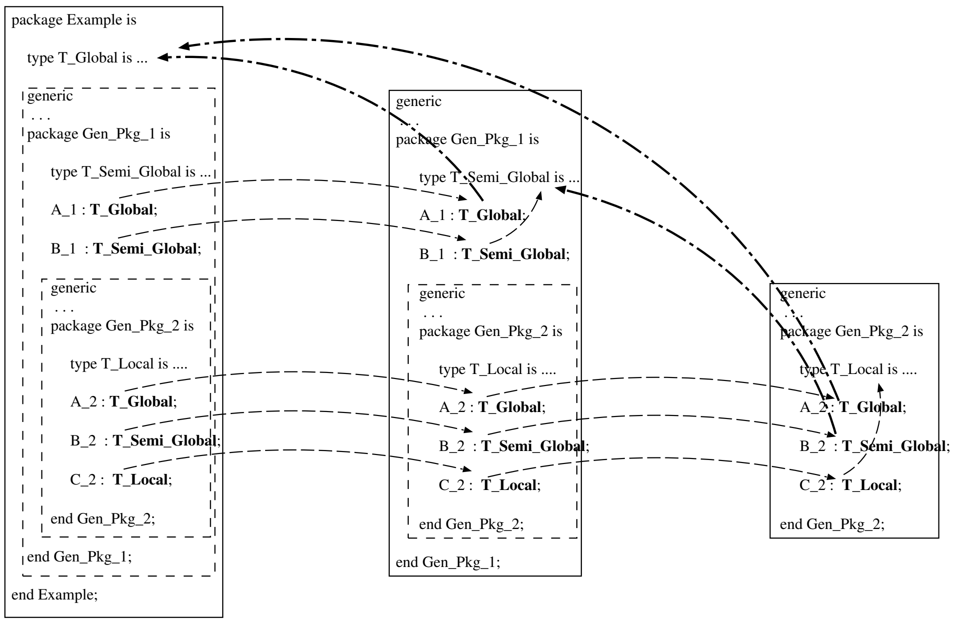


Figure 7.7: Analysis of nested generic package *Gen_Pkg_2*.

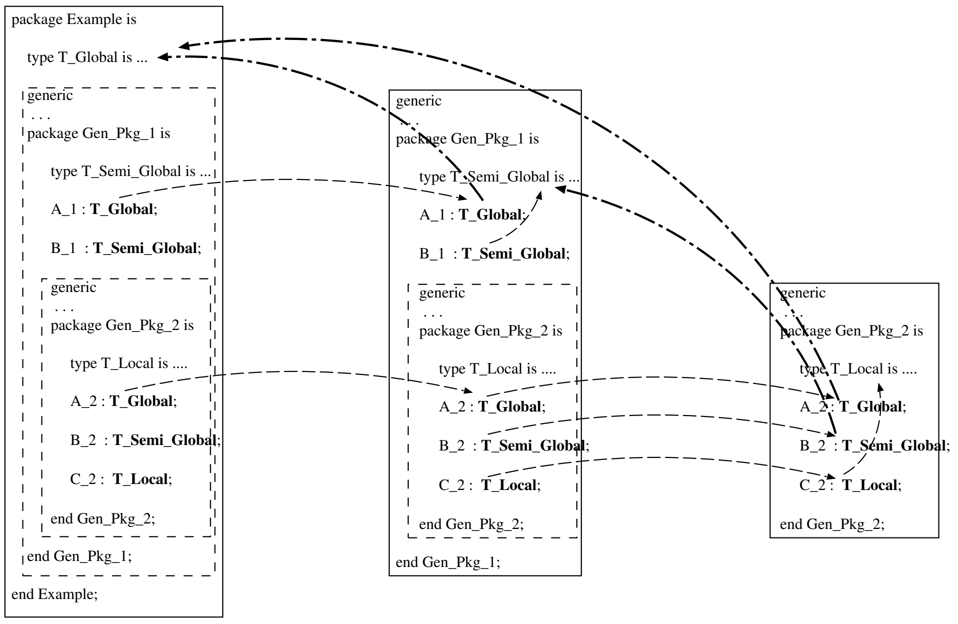
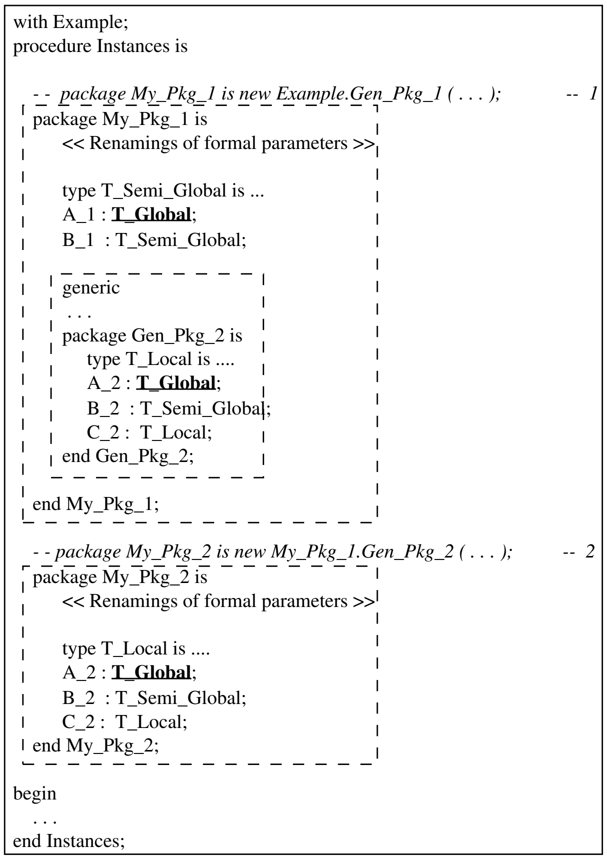


Figure 7.8: Saving global references.

first dotted rectangle is the instantiation of *My_Pkg_1*; the second dotted rectangle is the instantiation of the generic unit in the instance *My_Pkg_1*.



```
with Example;
procedure Instances is

  - -  package My_Pkg_1 is new Example.Gen_Pkg_1 ( . . . );          -- 1
  package My_Pkg_1 is
      << Renamings of formal parameters >>

      type T_Semi_Global is ...
      A_1 : T_Global;
      B_1  : T_Semi_Global;

      generic
       . . .
      package Gen_Pkg_2 is
          type T_Local is ....
          A_2 : T_Global;
          B_2  : T_Semi_Global;
          C_2 :  T_Local;
      end Gen_Pkg_2;

  end My_Pkg_1;

  - - package My_Pkg_2 is new My_Pkg_1.Gen_Pkg_2 ( . . . );          -- 2
  package My_Pkg_2 is
      << Renamings of formal parameters >>

      type T_Local is ....
      A_2 : T_Global;
      B_2  : T_Semi_Global;
      C_2 :  T_Local;
  end My_Pkg_2;

begin
  . . .
end Instances;
```

Figure 7.9: Instantiation of nested generic units.

7.3 Generic Child Units

7.3.1 Analysis of Generic Child Units

The analysis of generic child units adds no additional complexity to the mechanisms described in the previous sections. Conceptually the child unit has an implicit *with_clause* to its parents. Therefore the analysis of a generic child unit recursively loads and analyzes its parents. For example, to analyze *R* (child of *P.Q*) the front-end loads and analyzes *P*, then *Q* and finally *R*. This is identical to what is done for non-generic units. Of course, the analysis of the generics ends with the recovery of global names. References to entities declared in a generic ancestor are not preserved, because at instantiation time they will resolve to the corresponding entities declared in the instance of the ancestor.

7.3.2 Instantiation of Child Generic Units

Let us assume *P* is a generic library package, and *P.C* is a generic child package of *P*. Conceptually, an instance *I* of *P* is a package that contains a nested generic unit called *I.C*[AAR95, Section 10.1.1(19)]. A generic child unit can only be instantiated in the context of an instantiation of its parent, because of course it will in general depend on the actuals and the local entities of the parent. This clause of the RM simply indicates that a partially parameterized child generic unit is implicitly declared within the instantiation of the parent. Nevertheless, given that library packages are in general written in separate source files (this is mandatory in the GNAT compilation model [Dew94]), the compilation of *P* does not really contain information concerning *I.C*, and the front-end must handle generic child units with care to find the proper sources. Let us examine this mechanism in detail, and indicate how the proper visibility is established at the point of instantiation.

The RM paragraph just quoted allows generic child units to be instantiated both as local packages and as independent compilation units. A common idiom is to have an instantiation hierarchy that duplicates the generic hierarchy, but it is also possible to instantiate several members of a generic hierarchy within a common scope. Let us consider the following generic units in each case:

```
generic
...
package Parent is
   ...
end Parent;

generic
...
package Parent.Child is
    ...
end Parent.Child;
```

1. **Local Instantiation**. In the first case the instantiation of the generic hierarchy is done within the same declarative part. For example:

```
1: with Parent.Child;
2: procedure Example is
3:    package Parent_Instance is
4:       new Parent (...);
5:    package Child_Instance is
6:       new Parent_Instance.Child (...);
7:    ...
8: end Example;
```

First note that the name of the generic unit in the second instance (line 5) is that of the implicit child in the instance of the parent, and not that of generic child unit that was actually analyzed. The first step in processing the instantiation is to retrieve the name of the generic parent, and from there to recognize that the child unit that we are about to instantiate is *Parent.Child*.

To analyze the instantiation of *Parent.Child*, we must place the tree for this unit in the context of the instantiation of the *Parent*. This is so that a name in the child that resolved to an entity in the parent will now resolve to an entity in the instance of the parent. As a result, after producing the required copy of the analyzed generic unit for *Parent.Child*, the front-end places *Parent_Instance* on the scope stack. The analysis of the child instance will now resolve correctly. After analysis, the front-end removes *Parent_Instance* from the scope stack, to restore the proper environment for the rest of the declarative part.

2. **Library Level Instantiation**. This case is more complex than that of a local instantiation. Let us assume the following instantiation of the parent:

```
with Parent;
package Parent_Instance is new Parent (...);
```

The RM rule requires that the parent instance be visible at the point of instantiation of the child, so it must appear in a *with_clause* of the current unit, together with a *with_clause* for the generic child unit itself.

```
1: with Parent.Child;
2: with Parent_Instance;
3: package Child_Instance is new Parent_Instance.Child (...);
```

The *with_clause* for the child (line 1) makes the implicit child visible within the instantiation of the parent [Bar95, Section 10.1.3]. The instantiation at line 3 names this implicit child.

Using a similar approach to the previous case, the front-end retrieves the generic parent, verifies that a child unit of the given name is present in the context, and retrieves the analyzed generic unit. Its analysis must again be performed in the scope of the parent instance. Even though the parent instance was separately compiled, all semantic information is available because it appears in the context. Therefore the front-end places it on the scope stack (as before), and performs the analysis the child instance, after which the parent instance is unstacked.

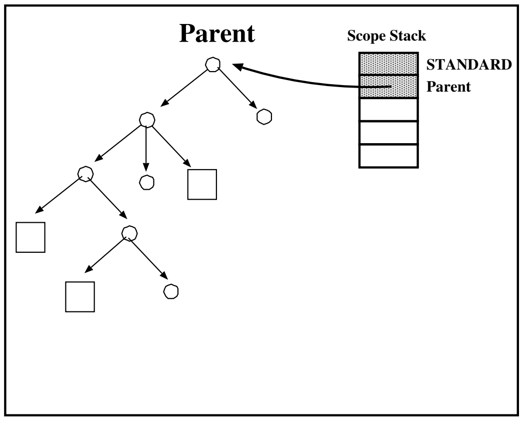
Clearly, more that one ancestor may be involved, and all of them have to be placed on the scope stack in the proper order before analyzing the child instance.

Figure 7.10 represents the sequence followed by the GNAT front-end to instantiate *Child_Instance* (according to our previous example):

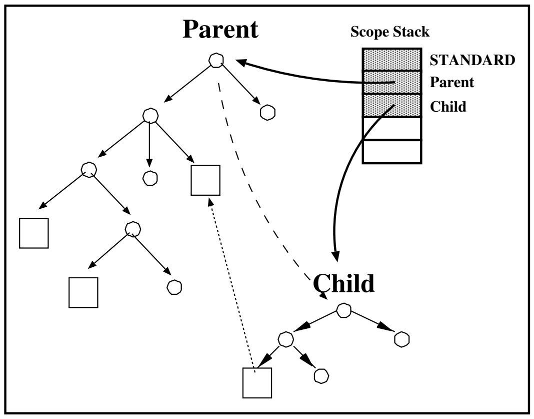
1. The upper two figures correspond to the analysis of line 1 (*with Parent.Child*). First, the GNAT front-end loads and analyzes *Parent* (Step 1.1). During this analysis the *Scope Stack* of the front-end provides visibility to all the entities in package Standard, plus the entities declared in *Parent* (see the *Scope Stack* in the left side of the figure). After the *Parent* unit has been analyzed, the front-end loads and analyzes its generic *Child* (Step 1.2). Here the *Scope Stack* of the front-end provides visibility to all the entities visible to its parent plus the entities declared in the child. When the generic child unit is analyzed the visibility to this hierarchy is removed from the *Scope Stack* and the front-end is ready to analyze the next context clause.

2. The figure in the middle (Step 2) corresponds to line 2. Because the instantiation of child generics requires the context of its parents, the analysis of the second *with_clause* forces a new instantiation of *Parent_Instance*; the AST of the generic parent is thus copied and analyzed in the global scope. In analogy to the previous case, after the instance is analyzed its visibility is removed from the Scope Stack.

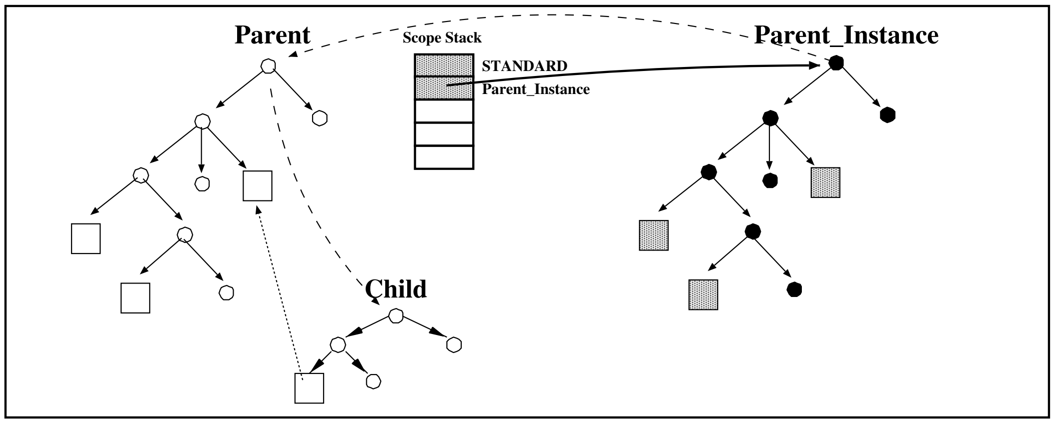
3. The figure in the bottom (Step 3) corresponds to the instantiation of the generic child (line 3). To provide the right visibility through the Scope Stack, the front-end (1) retrieves the generic parent of *Parent_Instance* (arrow from *Parent_Instance* to its generic *Parent*), (2) verifies that a child unit of *Parent* is present in the context (arrow from the generic *Parent* to its generic *Child*), (3) installs *Parent_Instance* in the Scope Stack. (4) makes a copy of the tree corresponding to the generic Child, (5) installs the child instance on the Scope Stack, and finally analyzes it.



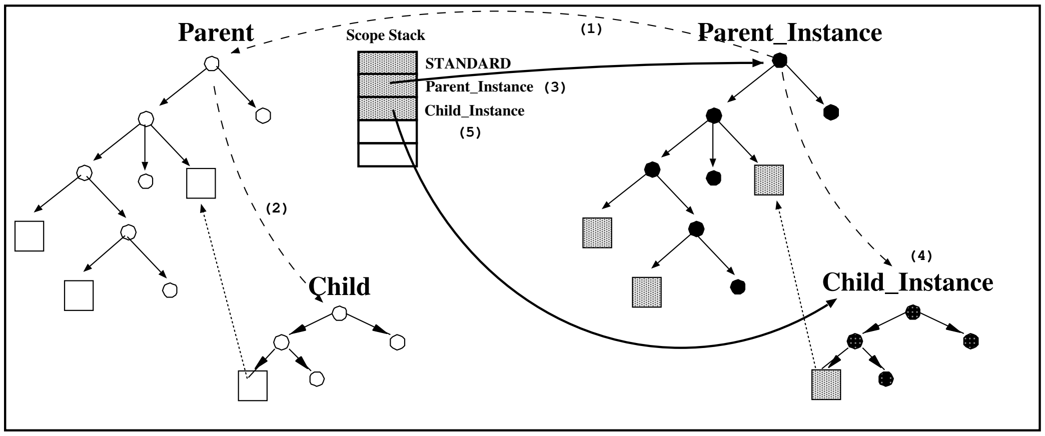
Step 1.1: Analyze Parent



Step 1.2: Analyze Parent.Child



Step 2: Analyze and Instantiate Parent



Step 3: Analyze and Instantiate Child

Figure 7.10: Sequence of steps to instantiate *Child_Instance*.

7.4 Delayed Instantiation of Bodies

The front-end instantiates generic bodies in a separate pass, after it completes all semantic analysis of the main unit. By delaying the instantiation of bodies, GNAT is able to compile legal programs that include cross-dependences among several package declarations and bodies. Such dependences present difficult problems in the order of elaboration to other compilers. For example, consider the following Ada code:

```
-- ---------------------------------------Specifications
package A is
   generic ...
   package G_A is ...
end A;

package B is
   generic ...
   package G_B is ...
end B;

-- -----------------------------------------------Bodies
with B;
package body A is
   package N_B is new B.GB (...);
end A;

with A;
package body B is
   package N_A is new A.G_A (...);
end B;
```

Conventional compilation schemes either reject these instantiations as circular (even though they are not) or are forced to use an indirect linking approach to the instances, which is inefficient. In GNAT, the instantiated bodies are placed in-line in the tree. This can only be done after regular semantic analysis of the main unit is completed. Up to that point, only the declarations of instances have been created and analyzed. It is worth noting that instantiations can appear before the corresponding generic body, which indicates that the body of the corresponding instance cannot be place at the same place as its specification, because it may depend on entities defined after the point of instantiation. In that case, a delicate analysis is required to locate the correct point at which an instance body must be inserted.

NOTE. The table *Pending_Instantiations* (cf. Package *Inline*) keeps track of delayed body instantiations. Inline handles the actual calls to do the body instantiations. This activity is part of *Inline*, since the processing occurs at the same point, and for essentially the same reason, as the handling of calls to inlined routines. Note that an instance body may contain other instances, and that the list of pending instantiations must be treated as a queue to which new elements are added on the fly.

7.5 Detection of Instantiation Circularities

A circular chain of instantiations is a static error which must be detected at compile time. The detection of these circularities is carried out at the point that the front-end creates a generic instance specification or body. The circularity is detected using one of several strategies. If there is a circularity between two compilation units, where each contains an instance of the other, then the analysis of the offending unit will eventually result in trying to load the same unit again, and the parser detects this problem as it analyzes the package instantiation for the second time. If a unit contains an instantiation of itself, the front-end detects the circularity by noticing that the scope stack currently holds another instance of the generic. If the circularity involves subunits, it is detected using a local data structure that lists, for each generic unit, the instances that it declares.

7.6 Summary

A generic unit is essentially a macro for a subprogram or package, whose parameters can be types, variables, subprograms and packages. Generic units must be instantiated, by suppling specific actuals for the parameters. This conceptually creates a copy of the specification and body which are appropriately customized.

There are two main implementation techniques for generic instantiation known as *Macro Expansion* and *Shared Generics*. GNAT implements generics by expansion. Because the GNAT compilation model generates neither code nor intermediate representation in case of generic specifications, they are analyzed each time they are found in an Ada context clause. This analysis involves the creation of a copy of the fragment of the AST corresponding to the generic specification. The front-end analyzes this copy and updates in the original fragment of the AST all the non-local references to entities (the modified original tree). For each instance of the generic unit the front-end generates a new copy of this modified fragment of the AST, adds some renaming declarations to handle actual parameters, and analyzes and expands this copy. The GNAT front-end incorporates extensions to this mechanism to analyze and instantiate nested generic units and generic child units.

Chapter 8

Freezing Analysis

Ada allows programmers to provide explicit information on the final representation of an entity [AAR95, Section 13]. The point at which the final representation is fully determined is called a *Freezing Point*. Obviously, if no representation is explicitly given the compiler has to take some default decision. When an entity is frozen it causes freezing of all depending entities. Following table summarizes the list of dependent entities to be frozen when a construct is frozen.

Construct being frozen	Causes freezing of
Static Expression	Entities in expression and their types, plus the type of the expression
Non-static Expression	Type of expression
Allocators	Designated subtype of qualified expression or subtype indication
Object	Corresponding subtype (except deferred constants)
Subtype	Corresponding type
Derived subtype	Parent subtype
Type with components	Component subtypes
Constrained Array	Index subtypes
Subprograms and entries	Parameter and result subtypes
Tagged type	Corresponding Class-wide type
Tagged type extension	Parent tagged type
Class-wide type	Corresponding Tagged type
Generic Instantiation	Parameter subtypes

GNAT generates a *Freeze_Entity* node at the point where an entity is frozen, and sets a pointer from the defining entity to its corresponding freeze node. Freeze nodes are later processed by the expander to generate associated code and data which must be delayed until the representation of the type is frozen (for example, an initialization subprogram). They are also used by Gigi to determine the point at which the object must be elaborated. If the GNAT compiler were fully faithful to the Ada model, it would generate freeze nodes for all entities, but that is a bit heavy. Therefore it freezes at their point of declaration all the entities whose representation characteristics can be determined at declaration time (cf. Package *Freeze*).

Although the Ada Reference Manual defines the freezing points of the language, the compiler must have special care with some details. These details are discussed in the following sections.

8.1 Freezing Types and Subtypes

All declared types have freeze nodes. This includes anonymous base types created for constrained type declarations, where the defining identifier is a first subtype of the anonymous type. An obvious exception to this rule are access types, because freezing an access type does not freeze its designated subtype. In addition, all first subtypes have freeze nodes. Other subtypes only need freeze nodes if the corresponding base type has not yet been frozen; if the base type has been frozen there is no need for a freeze node because no representation clauses can appear for the subtype in any case.

Types and subtypes are frozen by 1) object declarations, 2) bodies found in the same declarative region (for example, a subprogram body), and 3) the end of their corresponding declarative part. When a user-defined type is frozen generally forces freezing of all the entities on which it depends (default expressions have an special treatment, and will be described later). For example freezing a subtype definition forces freezing of all the derivations found from this derived type to the root type. Implicit types, types and subtypes created else by the semantic analyzer or by the expander to reflect the underlying Ada semantics, are frozen when the corresponding derived type is frozen. For example, constrained type definitions are internally converted by the GNAT compiler into an implicit *anonymous base type* definition plus a derived type definition:

```
type T is new Integer;
  -- Internally transformed by the compiler into:
  --     type TB is new Integer;
  --     subtype T is TB;

type T is array (1..10) of ...;
  -- Internally transformed by the compiler into:
  --     subtype TD is Integer;
  --     type TB is array (TD range <>) of ...;
  --     subtype T is TB (1..10);
```

In this example *TB* and *TD* are implicit types which will be frozen when their derived type *T* is frozen. Freezing of implicit types introduced by component declarations (i.e. component constraints) do not need to be delayed because the semantic analyzer verifies that the parents of the subtypes are frozen before the enclosing record is frozen (cf. Subprogram *Build_Discriminated_Subtype*).

8.2 Freezing Expressions

In general, naming an entity in any expression causes freezing of the corresponding type [AAR95, Section 13.14]. However, the compiler must have special care with default expressions used in discriminants and constraints of record components; they must be frozen at the point of the corresponding object declarations (not at the point of the type definition). For example:

```
type T_Int is new Integer;
type T_Name is array (T_Int range <>) of Character;

type T (Max : T_Int) is              -- Do not freeze T_Int here
   record
       Name   : T_Name (1 .. Max); -- Do not freeze T_Int here
       Length : T_Int := Max;       -- Do not freeze T_Int here
   end record;

My_Object : T (30);                  -- Freeze T_Int, T_Name, T
                                     -- and My_Object
```

Similarly, default expressions used in subprogram parameters must be deferred until the subprogram is frozen. In case of procedures this occurs when a body is found, but in case of functions the compiler must also consider function calls found in object initializations.

8.3 Freezing Objects

Simple objects are frozen at their point of declaration. Special care must be taken with deferred constants, which can not be frozen until the point of the full constant declaration. Objects with dynamic address clauses have a delayed freeze. In the freezing point Gigi generates code to evaluate the initialization expression (if present).

Protected types must be handled with special care because the protected-type definition is expanded into a record-type definition. For example:

```
protected type PO (<Discriminants>) is
   ...
private
   <Private_Data>; --  <1>
end PO;

--  Internally transformed by the GNAT compiler into:
--
--     type poV (Discriminants) is new Limited_Controlled with
--        record
--           <Private_Data> -- <2>
--           _object : Run_Time_Data_Type (<Num_Entries>);
--        end record;
--     ...
```

As a consequence, the subtypes of components of protected-type definitions (line 1) do not need freezing. Freezing actions correspond to the equivalent components in the record-type definition (line 2) and, according to the Ada freezing rules, this must be delayed until the point of the first object declaration of type *poV*.

8.4 Freezing Subprograms

Formal parameters are frozen when the associated subprogram specification is frozen, so there is never any need for them to have delayed freezing. Subprograms specifications are frozen at the point of declaration unless one or more of the formal types, or the return type, have delayed freezing and are not yet frozen. This includes the case of a formal access type where the designated type is not frozen. Subprogram bodies freeze everything.

8.5 Freezing Packages

Freezing rules for packages are quite clear. The language specifies that the end of a library package causes freezing of each entity declared within it, except for incomplete and private types which must be deferred until the full-type declaration is found. Special care must be taken with incomplete types because it is legal to place it in the package body [AAR95, Section 3.10.1(3)]. In addition, because the semantic analyzer keeps the entities of the partial and full views, it must also propagate the freezing information from the full view to the partial view (cf. *Find_Type_Name* subprogram).

In the case of local packages, the end of the package body causes the same effect of a subprogram body; it causes freezing of the entities declared before it within the same declarative part [AAR95, Section 13.14(3)].

8.6 Freezing Generic Units

Obviously, because no code is generated for a generic unit, generic units do not have freeze nodes; the freeze nodes must be generated in the instances. Library-level instances freeze everything at the end of the corresponding body. However, local instances only cause freezing of: (1) its real parameters as well as their default-expressions (if any), and (2) the entities declared before it within the same declarative part. This behavior avoids a premature freezing of global types used in the generic. In addition, the compiler must have special care with early instantiation of local packages. For example:

```
procedure Example is
   generic
   ...
   package G is
      ...
   end G;

   procedure Local is
      package I (...) is new G;   -- Early instantiation
                                  -- Freezing point (freeze i)
   begin
      ...
   end;
   package body G is
      ...
   end G;

begin
   ...
end Example;
```

At the point of the early instantiation, the Semantic Analyzer generates an instance of the specification (required to continue the analysis of the subprogram *Local*); the corresponding instance is generated in a latter pass (cf. described in Section 7.4). However, the semantic analyzer must add a freezing node to give GiGi the right order of elaboration of the packages (cf. *Install_Body* and *Freeze_Subprogram_Body* subprograms).

8.7 Summary

Ada allows programmers to provide explicit information on the final representation of an entity. The compiler generates a *Freeze_Entity* node at the point where an entity is frozen, and sets a pointer from the defining entity to its corresponding freeze node. Freeze nodes are later processed by the expander to generate associated code and data. Although the Ada Reference Manual defines the freezing points of the language, the compiler must have special care with 1) anonymous, implicit and incomplete types, 2) default expressions, 3) protected types, and 4) generic units. These details have been discussed in this chapter.

Part III

Third Part: Expansion

Chapter 9

Expansion of Tasks

A task type is a template from which actual task objects are created. A task object can be created either as part of the elaboration of an object declaration occurring immediately within some declarative region, or as part of the evaluation of an allocator (an expression in the form "**new**..."). The *Parent* is the task on which a task depends. If the task has been declared by means of an object declaration, its parent is the task which declared the task object; if the task has been declared by means of an allocator (an Ada expression in the form '**new** ...'), its parent is the task which has the corresponding access declaration. When a parent creates a new task, the parent's execution is suspended while it waits for the child to complete its activation (either immediately, if the child is created by an allocator, or after the elaboration of the associated declarative part). Once sevthe child has finished its activation, parent and child proceed concurrently. If a task creates another task during its own activation, then it must also wait for its child to activate before it can begin execution (cf. [BW98, Chapter 4.3.1] and [Bar99, Chapter 17.7]).

Conceptually every Ada program has a task (called the *Environment Task*) which is responsible for the program elaboration. The environment task is generally the operating system thread which initializes the run-time and executes the main Ada subprogram. Before calling the main procedure of the Ada program, the environment task elaborates all library units needed by the Ada main program. This elaboration causes library-level tasks to be created and activated before the main procedure starts execution.

The lifetime of a task object has three main phases: (1) *Activation*, elaboration of the declarative part of the task body. The *Activator* denotes the task which created and activated the task. All tasks created by the elaboration of object declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together [AAR95, Section 9]; (2) *Normal execution*, execution of the statements visible within the body of the task; and (3) *Finalization*, execution of any finalization code associated with the objects in its declarative part.

In order to provide this run-time semantics, the front-end performs substantial transformations on the AST, and adds numerous calls to run-time routines. Each tasking primitive (rendez-vous, selective wait, delay statement, abort, etc.) is expanded into generally target-independent code. The run-time information concerning each Ada task (task state, parent, activator, etc. [AAR95, Chapter 9]) is stored in a per-task record called the *Ada Task Control Block (ATCB)* (cf. Section 14.1. In the examples that follow, the interface to the run-time is described generically by the name GNARL, which in practice corresponds to a collection of run-time packages with the same interface, and bodies that are OS-dependent.

Details are provided in the following sections. The description unavoidably combines issues of run-time semantics with analysis and code expansion.

9.1 Task Creation

According to Ada semantics, all tasks created by the elaboration of object declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together [AAR95, Section 9.2(2)]. In addition, if an exception is raised in the elaboration of a declarative part, then any task T created during that elaboration becomes terminated and is never activated. As T itself cannot handle the exception, the language requires the parent (creator) task to deal with the situation: the predefined exception *Tasking_Error* is raised in the activating context.

In order to achieve this semantics, the GNAT run-time uses an auxiliary list (the *Activation List*). The front-end expands the object declaration by introducing a local variable that holds the the activation list, and splits the OS call to create the new thread into two separate calls to the GNAT run-time: (1) Create_Task, which creates and initializes the new ACTB, and inserts it into both the all-tasks and activation lists, and (2) Activate_Task, which traverses the activation list and activates the new threads.

9.2 Task Termination

The concept of a *master* is the basis of the rules for task termination in Ada (cf. [AAR95, Section 9.3]). A Master is a construct that performs finalization of local objects after it is complete, but before leaving its execution altogether. For finalization purposes, a master can be a task body, block statement, subprogram body i.e. any construct that contains local declarations. The language also considers accept statements as masters because the body of an accept may contain aggregates and function calls that create anonymous local objects that may require finalization. Task objects declared by the master, or denoted by objects of access types declared by the master, are said to depend on that master. Dependent tasks must terminate before a master performs finalization on other objects that it has created. Since accept statements have no declarative part, tasks are never dependent on them, and so a master for purposes of task termination is a task body, block statement, or subprogram body.

To properly implement task termination, the run-time must be notified of the change in master-nesting whenever execution is about to enter or leave a non-trivial master. Therefore, in this case the front-end generates calls to the run-time subprograms Enter_Master and Complete_Master (or, in the case of tasks, via Create_Task and Complete_Task subprograms).

9.3 Task Abortion

The abort statement is intended for use in response to those error conditions where recovery by the errant task is deemed to be impossible. Tasks which are aborted are said to become *abnormal*, and are thus prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately. However, certain actions must be protected in order that the integrity of the run-time be assured. For example, the following operations are defined to be abort-deferred [BW98, Section 10.2.1]: (1) a protected action, (2) waiting for an entry call to complete, (3) waiting for termination of dependent tasks, and (4) the execution of the controlled-types *initialize*, *finalize*, and *adjust* subprograms. This means that execution of abnormal tasks that are performing one of these operations must continue until the operation is complete.

For this purpose the GNAT run-time has a pair of subprograms (Abort_Defer, Abort_Undefer) which are used by the code expanded by the front-end for the task body (cf. Section 9.5), the rendezvous statements (cf. Chapter 10), and protected objects (cf. Chapter 11). Intuitively, Abort_Defer masks all interrupts until Abort_Undefer is invoked.

9.4 Expansion of Task Type Declarations

A task type declaration is expanded by the front-end into a limited record type declaration. For example, let us consider the following task specification:

```
task type T_Task (Discriminant : DType) is
   ...
end T_Task;
```

It is expanded by the front-end into the following code:

```
T_TaskE : aliased Boolean := False;
T_TaskZ : Size_Type       := GNARL.Unspecified_Size; |
                             Size_Type (Size_Expression);

type T_TaskV [ (Discriminant : DType) ] is limited record
  _Task_Id : System.Tasking.Task_Id;
  [ Entry_Family_Name : array (Bounds) of Void; ]
  [ _Priority  : Integer        := Priority_Expression;      ]

  [ _Size      : Size_Type      := Size_Expression;          ]
  [ _Task_Info : Task_Info_Type := Task_Info_Expression;     ]
  [ _Task_Name : Task_Image_Type : new String'(Task_Name); ]
end record;
```

The code between brackets is optional; its expansion depends on the Ada features appearing in the source code. The rest is needed at run-time for all tasks: let us examine this expansion in detail. First we find two variable declarations: (1) a boolean flag `E`, which will indicate if the body of the task has been elaborated, and (2) a variable which holds the task stack size (either the default value, *unspecified_size*, or the value set by means of a pragma *Storage_Size*).

Each task type is expanded by the front-end into a separate limited record `V`. If the task type has discriminants, this record type must include the same discriminants. The first field of the record contains the *Task_ID*[1] value (an access to the corresponding ATCB, cf. Section 14.1). If the task type has entry families, one

[1]*System.Tasking.Task_ID*

Entry_Family_Name component is present for each entry family in the task type definition; their bounds correspond to the bounds of the entry family (which may depend on discriminants). Since the run-time only needs such information for determining the entry index, their element type is void. Finally, the next three fields are present only if the corresponding Ada pragma is present in the task definition: pragmas *Storage_Size*, *Task_Info*, and *Task_Name*.

9.5 Task Body Expansion

The task body is expanded into a procedure. For example, let us consider the following task body:

```ada
task body T_Task is
   <Declarations>
begin
   <Statements>
end T_Task;
```

It is expanded by the front-end into the following code:

```ada
procedure T_TaskB (_Task : access T_TaskV) is
   Discriminant : Dtype renames _Task.Discriminant;

   procedure _Clean is
   begin
      GNARL.Abort_Defer;
      GNARL.Complete_Task;
      GNARL.Abort_Undefer;
   end _Clean;

begin
   GNARL.Abort_Undefer;
   <Declarations>
   GNARL.Complete_Activation;
   <Statements>
at end
   _Clean;
end T_TaskB;
```

The procedure receives a single parameter *_Task*, which is an access to the corresponding high-level record (also generated by the expander, cf. section 9.4).

This parameter gives access to the discriminants of the task object, which can be used in the code of the body. The renaming (line 2) simplifies the access to the discriminants in the expanded code. The local subprogram *_Clean* is a common point of finalization; by means of the special *at end* statement, it is called at the end of the subprogram execution whether the operation completes successfully or abnormally.

Once the task is activated, it first executes code that elaborates its local objects (line 13), and then calls a run-time subprogram (Complete_Activation) to notify the activator that the elaboration was successfully completed. At this point the task becomes blocked until all tasks created by the activator complete their elaborations; if any of them fails, the task is aborted and *Tasking_Error* is raised in the activator. If the task is not aborted, it is allowed to proceed and execute its statements (line 15) under control of the run-time scheduler. When the task terminates (successfully or abnormally), the local subprogram *_Clean* is executed, which calls the run-time to perform its finalization. Even though a completed task cannot execute any longer, it is not yet safe to deallocate its working storage at this point because some reference to the task may exist in other tasks. In particular, it is possible for other tasks to still attempt entry calls to a terminated task, to abort it, and to interrogate its status via the *'Terminated* and *'Callable* attributes. For this reason, the resources are not deallocated until the master associated with the task completes. In general this is the earliest point at which it is completely safe to discard all storage associated with dependent tasks, because it is at this point that execution leaves the scope of the task's type declaration, and there is no longer any way to refer to the task. Any references to the task that may have been passed far from its point of creation, as via access variables or functions returning task values [BR85, Section 4] are themselves dead at this point.

9.6 Example of Task Expansion

To help the reader to understand the sequence of run-time actions involved in the life-time of Ada tasks, let us summarize the code expansion presented in Sections 9.1, 9.4, and 9.5 by means of a simple example. Let us consider the following Ada code:

```
procedure Activator is
   task My_Task;
   task body My_Task is
      < Local Declarations >
   begin
      < Task body statements >
   end My_Task;
begin
   < Additional Statements >
end Activator;
```

This code is expanded by the GNAT front-end as follows:

```
procedure Activator is

   My_TaskE : aliased Boolean := False;
   My_TaskZ : Size_Type       := GNARL.Unspecified_Size;
   type My_TaskV is limited record
       _Task_Id : System.Tasking.Task_Id;
   end record;

   procedure My_TaskB (_Task : access T_TaskV) is
      procedure _Clean is
      begin
         GNARL.Complete_Task;                              -- (6)
      end _Clean;
    begin
      << Expanded code to elaborate local declarations >>
      GNARL.Complete_Activation;                           -- (4)
      < Task body statements >                             -- (5')
   at end
     _Clean;
   end My_TaskB;

   My_Task   : My_TaskV;
   _Chain    : GNARL.Activation_Chain;
begin
   GNARL.Enter_Master;                                     -- (1)
   GNARL.Create_Task                                       -- (2)
      (My_Task, My_TaskZ, My_TaskB'Access, _Chain, ...);
   GNARL.Activate_Task (_Chain);                           -- (3)
   < Additional Statements >                               -- (5')
at end
   GNARL.Complete_Master;                                  -- (7)
end Activator;
```

The numbers in the comments to the right of the code present the execution sequence. First, because the main procedure has a task object declaration, it notifies the run-time that it is executing a master scope (step 1). It then creates the task ATCB (step 2), and activates the corresponding thread (step 3). After the task completes the elaboration of its local objects, it calls the run-time to report that it has completed its activation (step 4). From here on the execution of the task body and the main subprogram proceed in parallel (steps 5'). When the task terminates it notifies the run-time of its termination (step 6). When the body of the activator completes, it calls the run-time to wait for dependent tasks that may not have completed (step 7). Once it is established that the dependent task has terminated, the run-time recovers the task resources, and leaves the activator to terminate. If the activator calls Complete_Master before the dependent task completes its execution, the activator is blocked by the run-time until the dependent task body notifies its termination.

9.7 Summary

In this chapter we have seen the basic data structures used to support Ada tasks, and the corresponding task expansion. The run-time associates to each task an Ada Task Control Block (ATCB). Although the run-time registers the ATCBs in a linked list, one auxiliary list is required to implement the Ada semantics for tasks activation. Therefore, the front-end generates a temporal variable used to reference the elements in this list, and the run-time calls to create and activate the tasks. The front-end also generates calls to the run-time at the points at which the user code enters or leaves a master scope.

The Ada task specification is expanded into a limited record. The Ada task body is expanded into a procedure with calls to the run-time to notify: the successfully task activation, and the task termination.

Chapter 10

Expansion of Rendezvous and related Constructs

The *Rendezvous* is the basic mechanism for synchronization and communication of Ada tasks. Task communication is based on a client/server model of interaction. One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients). It does this by declaring one or more public *entries* in its task specification. A rendezvous is requested by one task by means an entry call on an entry of another task. For the rendezvous to take place the called task must accept this entry call. During the rendezvous the calling task waits while the accepting task executes. When the accepting task completes the request, the rendezvous ends and both tasks are freed to continue their execution asynchronously.

The parameter profile of entries is the same as that of Ada procedures (**in**, **out** and **in out**, with default parameters allowed for **in** parameters). Access parameters are not permitted, though parameters of any access type are, of course, allowed. Entries are overloadable entities. In addition, a task can have entry families (basically an array of entries). At run-time, each entry is associated with a queue of entry calls. Each entry queue has an attribute associated with it, the 'Count attribute. The task that declares an entry can use this attribute to determine the number of callers awaiting service on this entry.

Ada defines four entry-call modes: *simple, conditional, timed*, and *asynchronous* [AAR95, Section 9.5.3]. A simple mode entry-call is much like a procedure call; it may have parameters, which permit values to be passed in both directions between the calling and accepting tasks. Semantically the calling task is blocked until completion of the requested rendezvous. If the call is completed normally, the caller resumes execution with the statement following the call, just as it would after return from a procedure call. Recovery from any exception raised by the call is also treated as it would be for a procedure call. One minor difference detectable by the calling task is that an entry call may result in *Tasking_Error* being raised in the calling task, whereas an ordinary procedure call would not. The conditional and timed entry-calls allow the client task to withdraw the offer to communicate if the server task is not prepared to accept the call immediately or if does not accept the call within the stated delay, respectively. The asynchronous entry-call provides asynchronous transfer of control upon completion of an entry call. Similarly, on the acceptor task side there are also simple, conditional and timed modes.

10.1 Entry Identification

The run-time needs to uniquely identify each entry. For this purpose, the front-end associates each entry an numeric id, which a positive number which corresponds with the position of the entry in the task type specification. The following example shows this mapping.

```
task T is

   -- a simple entry
   entry Init (x : integer);

   -- An entry family with 3 entries has three different queues.
   entry Lock (1 .. 3) (Reason : in ...);
      -- Do_Work (1): Id = 2
      -- Do_Work (2): Id = 3
      -- Do_Work (3): Id = 4

   -- A simple entry declaration
   entry Unlock; -- Id = 5

end T;
```

10.2 Entry-Call Expansion

The entry call must communicate parameters between the caller and the server. Given that each task has its own stack, and in general the client cannot write on the stack of the server, this communication involves copying steps and indirection. The caller creates a parameter block that holds the actuals, and passes to the server a pointer to this block. An entry call also generates pre and post-actions, similar to those that are generated for procedure calls, to handle initialization of in-parameters, and creation of temporaries and copy-back for out and in-out parameters. The run-time structure associated with the call is the entry-call record (cf. described in Section 15.1). On the server side, the run-time places the pointer to the parameters block into the Uninterpreted_Data component of the entry-call record (See Expand_Identifier in Sem_Ch2). Figure 10.1 displays the data structures involved in a call to the following entry:

```
task T is
   entry E (Number : in Integer; Text : in String);
end T;
```

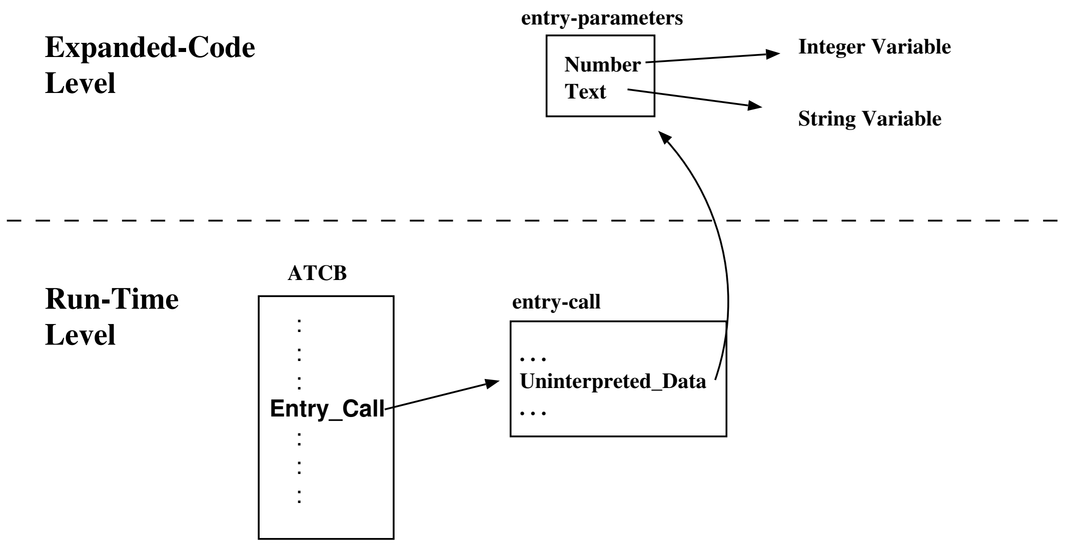


Figure 10.1: Data structures associated to an entry-call.

The following sections describe the expansion of each kind of entry-call : *simple, conditional, timed*, and *asynchronous*.

10.2.1 Expansion of Simple Entry-Calls

The front-end expands a simple mode entry call as follows:

```
declare
   type Params_Block is
      record
         Parm1 : Access_Param1_Type;
         ...
         ParmN : Access_ParamN_Type;
      end record;
   P : Params_Block := (Parm1'Access, ... , ParmN'Access);
begin
   GNARL.Call_Simple (Task_ID, Entry_ID, P'Address);
   [ Parm1 := P.Parm1; ]
   [ Parm2 := P.Parm2; ]
   [ ...               ]
end;
```

The address of the parameters record *P* is passed to the GNAT run-time along with the identifiers of the called task and entry. The assignments after the run-time call are present only in the case of **in-out** or **out** parameters for scalar types, and are used to copy back the resulting values of such parameters.

10.2.2 Expansion of Conditional and Timed Entry-Calls

A conditional entry call differs from a simple entry call in that the calling task will not wait unless the call can be accepted immediately. If the called task is ready to accept, execution proceeds as for a simple mode call. Otherwise, the calling task resumes execution without completion of a rendezvous. Recall the syntax for a conditional entry-call:

```
select
   entry-call
   <statements-1>
else
   <statements-2>
end select;
```

As the reader can see, other statements can appear after the entry-call, which are only executed if the call was accepted. The alternative branch can also include statements that are executed only if the caller is not ready to accept. The conditional entry-call is expanded as follows:

```
declare
   type Params_Block is record
      Parm1 : Access_Param1_Type;
      ...
      ParmN : Access_ParamN_Type;
   end record;

   P : Parms_Block := (Parm1'Access, ... , ParmN'Access);
   Successful : Boolean;
begin
   GNARL.Task_Entry_Call (Task_ID,
                          Entry_ID,
                          P'Address,
                          Successful);
   if Successful then
      [ Parm1 := P.Parm1; ]
      [ Parm2 := P.Parm2; ]
      [ ...               ]
      <statements-1>; -- Statements after the entry call
   else
      <statements-2>; -- Statements in the "else" part
   end if;
end;
```

In this case, the call to the run-time has an additional parameter (*Successful*) which indicates to the caller whether the entry-call was immediately accepted or not. If yes, the caller behaves as in the simple-mode case and assigns back the resulting values of the out-mode parameters (if present). Otherwise the caller executes the statements in the *else* part of the conditional entry-call.

The timed task entry call is handled by the GNAT compiler in a similar way. The main difference is the called run-time subprogram, because in this case if the entry-call can not be immediately accepted, the run-time must arm a timer and block the caller until the entry-call is accepted or else the timeout expires.

10.3 Asynchronous Transfer of Control

The Asynchronous Transfer of Control (ATC) allows the caller to continue executing some code while the entry call is waiting to be attended. Its Ada syntax is

[AAR95, Section 9.7.4]:

```
select
   triggering_alternative
then abort
   abortable_part
end select;
```

The triggering statement can be an entry-call or a delay-statement. If the triggering statement is queued the abortable part starts its concurrent execution. When one of the parts finishes its execution it aborts the execution of the other part. Thus the ATC provides local abortion which is potentially cheaper than the abortion of the entire task.

10.3.1 ATC Implementation Models

There are two implementation models for the ATC: the one-thread model, and the two-threads model. In the following description we will assume that the triggering statement is an entry-call statement (the case of the delay statement is similar).

- *One-Thread model.* In this model, the first action of the caller is to try to execute the entry-call. If the called task was ready to accept the entry-call then the abortable part is not executed; otherwise the run-time leaves the entry-call queued and leaves the caller to execute the abortable part. If the entry-call is accepted before the caller completes the abortable part, the caller is forced to transfer control to the triggered statements. On the contrary, the caller cancels the queued entry call.
- *Two-Threads model.* In the two-thread model, the task executing the ATC creates an agent-thread to execute the abortable part. Each thread just tries to execute its part and abort the other thread. The thread which completes its part wins!.

Proponents of the two-thread model argue that simplifies the implementation of several run-time aspects. One is that it preservers two useful invariants of the original Ada tasking model namely: (1) a thread that is waiting for an event is not executing, and (2) a thread never waits for more than a timeout and one other event. Another simplification is that the two thread model eliminates the need for one thread to asynchronously modify another thread's flow of control, which is not possible in some execution environments [GB94, Section 3.1]. However, the two-thread model seems to complicate the implementation at least as much as it simplifies it, and also violates a key invariant of Ada tasking: there is no longer a one-to-one correspondence between tasks and threads of control. This assumption pervades the semantics, and is the foundation of existing Ada tasking implementations. Loss of this invariant has many ramifications. Among these, data that previously could only be accessed by one thread of control becomes susceptible to race conditions. Thus, there are new requirements for synchronization, and new potential for deadlock within a single task. Also, just killing the agent thread is not as simple a solution as it might seem. There remains the problem of how to execute the agent's finalization code (if required due to the use of controlled types). If the operation that kills a thread does not support finalization, some other thread must perform the finalization. To do so, it must wait for the killed thread to be terminated to be able to obtain access to the run-time stack of the terminated thread. The latter may not be possible in systems where killing a thread also releases its stack space [GB94, Section 3.1].

By contrast, proponents of the one-thread model argue that it can be implemented with a signal and *longjmp()*. The triggering entry-call is just left queued while the abortable part is executed. If the abortable part completes first, the entry-call is removed. If the entry-call completes, the run-time sends an abortion signal to the caller. The signal handler for the abortion signal then transfers control out of the abortable part into the point of the entry-call [GMB93, Section 4.3].

Due to the disadvantages of the two-threads model, as well as the simplicity of the one-thread model, the GNAT compiler implements the one-thread model.

10.3.2 Expansion of ATC

The Following describes the expansion of an ATC statement:

```
 1: declare
 2:    P : aliased Parms := (Parm1'Access,..., ParmN'Access);
 3:    Successful : Boolean;
 4: begin
 5:    GNARL.Defer_Abortion;
 6:    GNARL.Task_Entry_Call (Task_ID, Entry_ID,
                              P'Access, Successful);
 7:    begin -- Abortable Part Scope
 8:       begin
 9:          GNARL.Undefer_Abortion;
10:          << Abortable Part >>
11:       at end
12:          GNARL.Entry_Call_Cancellation (Successful);
13:       end;
14:    exception
15:       when Abort_Signal =>
16:          GNARL.Undefer_Abortion;
17:    end;
18:    if not Successful then
19:       [ Parm1 := P.Parm1; ]
20:       [ Parm2 := P.Parm2; ]
21:       [ ...               ]
22:       << Triggered Statements >>
23:    end if;
24: end;
```

The first action issued in the scope associated with the ATC is to protect the entry call from abortion (line 5). From here two scenarios must be analyzed:

1. If the entry call is immediately accepted, the run-time subprogram called at line 6 completes the rendezvous, sets the Successful variable, and sends the abort signal to the caller. Because the abortion is deferred from line 5 onwards, this has no immediate effect. After the entry-call is completed, the caller now undefers the abortion (line 9), which raises the deferred abort signal and forces the caller to skip the abortable part and try to cancel the entry call (line 12). However, because the entry-call was completed this call does nothing. Finally, the caller assigns back the resulting values of the out-mode parameters, and executes the triggered statements (lines 18 to 24).

2. If the entry call is not immediately accepted, the caller undefers the abortion

(line 9) and executes the abortable part (line 10). Again here we have two possible scenarios:

(a) If the execution of the abortable part completes, the entry call is cancelled (line 12). The run-time sets Successful to false, to force the caller to skip lines 18 to 24.

(b) If the entry-call is completed before the abortable part completes, then the run-time sends the abort signal to the caller (which was executing the abortable part, line 10). This signal cancels the execution of the abortable part. The caller now tries to cancel the entry call (line 12), but because the entry-call was completed the only effect of this call is to set Successful to true, which forces the caller to execute the statements that assign back the resulting values of the out-mode parameters, and then execute the triggered statements (lines 18 to 24).

10.4 Expansion of Accept Statements

This section describes the expansion of the the simple and selective accept statements.

10.4.1 Simple Accept Expansion

Recall the syntax of simple accept statements:

```
accept E ( ... ) do
   << Entry Body Statements >>
end E;
```

A simple accept is expanded as follows:

```
declare
   Params_Block_Address : Address;
begin
   GNARL.Accept_Call (Entry_ID, Params_Block_Address);
   << Entry Body Statements >>
   GNARL.Complete_Rendezvous;
exception
   when others =>
```

```
          GNARL.Exceptional_Complete_Rendezvous;
end;
```

The acceptor task calls the run-time, specifying the identifier of the accepted entry, and receives the address of the parameters block to be used in the entry body statements. There are two different run-time subprograms which are called depending on whether the entry completes successfully or not: *Complete_Rendezvous* and *Exceptional_Complete_Rendezvous* respectively.

10.4.2 Timed and Selective Accept

The scheme used for expansion of the Ada timed calls and selective accept statements is similar. The only difference is the run-time subprogram that is invoked. In both cases the run-time receives from the caller a vector that indicates which entries are currently open: the *open-accept vector*. Each element of this vector has two fields: the entry identifier, and a boolean which indicates if the accept statement has a null body. Each element of the open-accept vector corresponds to the accept alternatives of the select statement; If the entry guard of a given alternative is closed, the corresponding entry identifier is set to 0. Consider the following example:

```
task T is
   entry P;                  -- Entry Id = 1
   entry Q;                  -- Entry Id = 2
end T;

task body T is
begin
   select
      accept Q do           -- OAV (1).Entry_Id   := 2
        << User Code >>     -- OAV (1).Null_Body  := False;
      end Q;
   or
      when <<User-Guard>> =>
               -- If the guard is open OAV (2).Entry_Id := 1
      accept P; --                else    OAV (2).Entry_Id := 0;
               -- OAV (2).Null_Body := True;
   else
      << else statements >>
   end select;
end T;
```

The GNAT front-end expands the selective accept into a block containing three declarations: the open-accept vector (OAV), the index of the selected alternative, and the address of the parameters block. A value of 0 in the index parameter is used by the run-time to indicate that the *else* alternative has been selected. Let us see a simplified version of the expansion of the previous example:

```
 1: declare
 2:    function P_Guard return Natural is
 3:    begin
 4:      if <<User-Guard>> then
 5:         return 1;  -- returns the Entry_Id
 6:      else
 7:         return 0;  -- return 0 (it is closed)
 8:      end if;
 9:    end P_Guard;
10:
11:    procedure Q_Body is
12:    begin
13:       GNARL.Undefer_Abortion;
14:       ... << User Code >>
15:       GNARL.Complete_Rendezvous;
16:    exception
17:       when others =>
18:          GNARL.Exceptional_Complete_Rendezvous;
19:    end Q_Body;
20:
21:    OAV     : GNARL.Open_Accepts_Table
22:               := ((2, True), (P_Guard, False), (0, False));
23:    Index : Natural;
24:    Params_Block_Address : System.Address;
25:
26:  begin
27:     GNARL.Selective_Wait
28:        (OAV, Params_Block_Address, Index);
29:     case Index is
30:        when 0 =>
31:           << else statements >>
32:        when 1 =>
33:           Q_Body;
34:        when 2 =>
35:           null;
36:     end case;
37:  end;
```

For each user-defined guard, the expander generates a function which evaluates the guard (lines 2 to 9): if the guard is open, this function returns the identifier of the entry; if the guard is close it returns 0. The entry-body statements are expanded inside local procedures following a scheme similar to the scheme described in Section 10.4.1 (lines 11 to 19). In addition, the front-end generates the open-accept vector with the corresponding initialization (lines 21-22), an *Index* variable used by the run-time to notify the selected alternative (line 23), and another variable used by the run-time to notify the address of the parameters block (line 24). After the call to the run-time (line 27), the expander generates a case-statement (lines 29 to 36) which uses the index returned by the run-time to execute the code associated to the alternative selected by the run-time.

10.4.3 Count Attribute Expansion

The 'Count attribute is expanded into a call to a run-time function (*Task_Count*) which receives as input parameter the identifier of the entry.

10.5 Summary

The rendezvous is the basic mechanism for synchronization and communication of Ada tasks. At the point of an entry-call, the front-end expands the actual parameters into a block with collects their addresses. After the call, the expander generates statements to copy the value of the out-mode parameters into the corresponding variables. In case of conditional and timed entry calls, the run-time returns one value which indicates the alternative to be executed; therefore it is also responsibility of the front-end to generate an if-statement to execute the right code. The expansion scheme followed in the implementation of ATC is similar to the conditional entry-call, although additional scopes must be generated to handle the abortion of the call.

For the implementation of the timed and selective accept statements the compiler expands: 1) each entry-guard into a function which evaluates the guard, and 2) each entry-body into a procedure which notifies the run-time if it was executed successfully or not. The expander generates code which collects all this information into an open-accept vector which passes to the run-time at the point of the call. In addition, after the call the run-time returns a value which indicates the identifier of the selected alternative, and the expander generates a case alternative which uses this value to execute the corresponding statements.

Chapter 11

Expansion of Protected Objects

Protected objects are an embodiment of the venerable notion of Monitor: a construct that allows shared data to be accessed by different threads under mutual exclusion. Protected objects provide data-driven synchronization between cooperating tasks: tasks communicate not only through rendez-vous or generally unsafe shared memory, but by disciplined access to shared objects with locks.

A protected object (PO) encapsulates data items, and allows their exclusive access and update by means of protected subprograms or protected entries. A protected function accesses the data in read-only mode, while a protected procedure has access in read-write mode. A protected entry is akin to a protected procedure, but it has a *Barrier* , that is to say a boolean expression that serves as a guard to the entry. The barrier provides a conditional version of the entry call: a calling thread has access to the data only if the barrier is True, otherwise the caller queues on the object until the barrier value becomes true, and no other task is currently active inside the protected object.

The declaration of a protected type comprises a visible interface and a private part, The visible interface includes only its operations (subprograms and entries). The private part describes the structure of the protected data.

The body of a protected type contains the bodies of the all the visible operations. It may also contain private operations. It does not contain any other declarations, to insure that the state of the object is fully described by the private part of the type declaration itself.

At run-time, the protected object is represented by a record that holds the protected data, a lock that insures mutual exclusion, and queues that hold blocked tasks. Each entry has its own queue, to hold tasks that await an opening of the corresponding barrier. The object itself has one queue that holds tasks competing for access to the object.

A call to a protected operation is similar to a call to a task entry. As with task entry-calls, the caller uses a selected component notation to specify the target of the call (task or protected object) and the operation to invoke. The call can be *simple*, *conditional*, or *timed*. Note however the critical distinction between task entry calls and protected entry calls: in the first case, the callee will execute the desired operation; in the second case, the caller task executes the operation, because the protected object is a passive structure with no thread of control.

After a protected operation is executed, the state of the object may have been affected, and the values of the barriers may have changed. It is therefore necessary to re-evaluate the barriers to determine whether some queued task can now have access to the object. This re-evaluation must be performed by the task that currently holds the lock on the object, that is to say the task that just completed executing a protected call. This means that if there are tasks queued on entries and tasks queued ”outside” of the object (on the outer queue for protected subprograms), those queued on entries will have priority. This is often explained in terms of the *eggshell model*. The object has an external shell that allows only one task at a time to proceed. The entry queues are all inside the shell, and they can hold any number of blocked tasks. The clean-up that follows the completion of an operation only concerns the tasks inside the shell.

As with tasks, protected objects may have private entries and families of entries. Private entries are not directly visible to users of the protected object; however they have their own queues, and are typically used in requeue operations. Entry families are arrays of entries, that is to say at run-time they correspond to arrays of independent queues. In a task body, An accept statement for a member of an entry family task specifies by means of an expression the member of the family being accepted. This means that different accept statements can be provided for different members of the family. By contrast, in a protected body there is a single entry body for the family. The index plays the role of an additional parameter of the entry body. The barrier of the entry body can use the index of the family (see examples of usage in [BW98, Chapter 7.5] and [Bar99, Chapter 18.9]). The attribute *Count* can be applied to protected entries, to provide the current number of tasks queued on the specified entry.

Given that the lifetime of a protected object and that of the tasks that use is, are not necessarily the same, it is possible for a protected object to disappear while some tasks are still queued on it (for example, the object may have been created dynamically, and explicitly deallocated). In that case the language semantics prescribes that the exception Program_Error must be raised on all queued tasks. This clean-up operation is most simply realized by implementing protected objects as controlled types. The Finalize procedure for protected types traverses all queues of the corresponding object and raises Program_Error on all tasks therein.

In order to understand the expansion activities associated with protected types, we will examine in some details the implementation model. After a discussion of run-time issues, we present the expansion of protected-type specifications, protected subprograms, barriers, and entry bodies. The expansion of protected entry calls is not discussed here because it is similar to that for task entry-calls (cf. Chapter 10).

11.1 The Proxy Model

As mentioned above, conceptually each protected operation is executed by a calling task. However, the functioning of a protected object includes house-keeping activities that also require execution, in particular the evaluation of barriers, and the language does not specify what thread is to compute them. The semantics specify that upon completion of a protected operation that may affect the state of the object, the barriers are evaluated at once to determine whether some queued task can gain access to the object, so that in principle a context-switch might take place at that point. However, if several tasks are now eligible to enter the object, each one of them will have to wait its turn to obtain the object, and a number of context-switches will be necessary to empty the queues. This suggests a different implementation, in which the task that completes its operation retains the lock on the object, and executes the entry bodies of the waiting tasks, *on their behalf*. In this fashion no additional context switches are needed. As far as the eligible tasks are concerned, their calls were executed and they can proceed. The first implementation model, in which each task executes the operation it invokes, is called the *Self-Service* model. The second one is called and the *Proxy* model.

The main advantages of the self-service model are that it permits more parallelism (on multiprocessor systems) and simplifies schedulability analysis. Parallelism is increased because on a multiprocessor system the exiting task can proceed with its own execution, in parallel with the execution of the next queued call. Schedulability analysis is simplified because a thread is allowed to continue with its own execution immediately after the (presumably bounded) time it takes to complete the body of the called protected operation and transfer the lock ownership to the next queued caller.

By contrast, the principal advantage of the proxy model is simplicity. If an entry body cannot be executed immediately, the calling task just suspends its execution; some other task will execute its entry-call. Complex features of protected objects, including timed and asynchronous entry-calls, are simplified even more by this model. As indicated above, on a uniprocessor the proxy model is more efficient because the number of context-switches is smaller. However, the proxy model compromises schedulability analysis, since the time to complete an operation is not bounded by the operation itself: it depends on the number of other tasks that may be queued on the object, and there is no upper-bound on the number of such calls that may be pending.

GNAT implements the proxy model because the implementation of the self-service model with Pthreads currently introduces crippling inefficiencies. The self-service model works best if the task attempting to leave the protected object can transfer directly the lock to a specific task that is waiting on an *Open* entry. However, there is no efficient way to achieve this under Pthreads. The existing mechanism requires raising the priority of the chosen task above that of all other contenders, and then rescanning the set of ready tasks to determine the one that is to be given access. This turns out to be unacceptably cosly.

11.1.1 Implementation

There are two possible implementations of the proxy model itself: *call-back* and *in-line*. In the call-back implementation the compiler transforms barriers and entry-bodies into stand-alone subprograms, and generates code to pass their addresses to the run-time; a single routine in the run-time implements the algorithm that calls these subprograms (cf. Figure 11.1). By constrast, in the in-line implementation the compiler not only expands the barriers and entry-bodies (and not necessarily inside subprograms), but also generates in-line the statements that implement the egg-shell model; after the execution of a protected-procedure or protected-entry, these statements reevaluate the barriers and execute the entry-body of the open entries until no candidate is selectable (cf. Figure 11.2).

The GNAT compiler uses the call-back implementation. The reasons are: 1) The call-back interface allows for much simpler translations, and eliminates some of the overhead inherent in the in-line interface's frequent alternation between the GNU Ada Run-Time and the application code. 2) The call-back interface has a big advantage in the simplicity and understandability of both the generated code and the internal logic of the compiler [GB95].

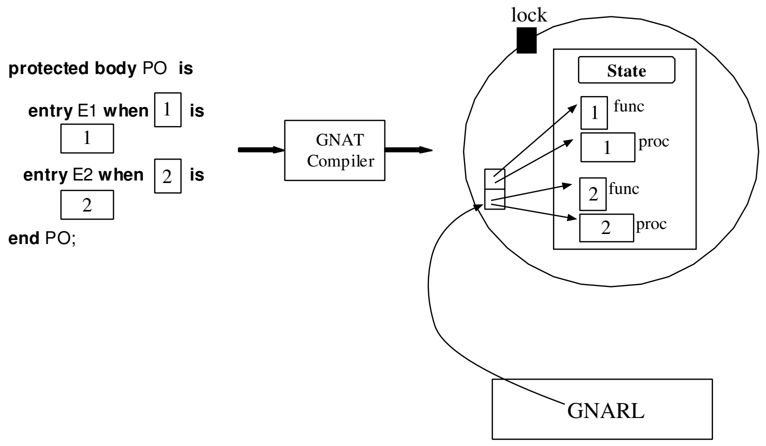


Figure 11.1: Proxy Model: Call-Back Implementation.

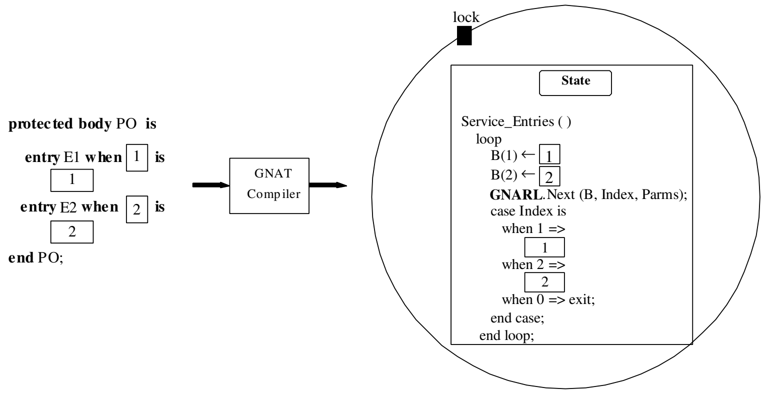


Figure 11.2: Proxy Model: In-Line Implementation.

11.2 Expansion of Protected Types

11.2.1 Expansion of Protected-Type Specification

The expansion of a protected type must create the following structures:

1. A record type that holds the protected data and the entry queues.
2. subprograms that implements each protected operation. The parameter list of each such subprogram includes a parameter that designates the object on which the operation is performed at run-time.

Consider the following protected type declaration:

```
protected type PO (Disc : Integer) is
   procedure P (C : Character);
   function  F (X : Integer) return Integer;
   entry E1;
   entry E2 (1..10)(X : Integer);
private
   Value : Integer;
end PO;
```

The front-end expands it as follows:

```
 1: type poV (Disc : Integer) is new Limited_Controlled with
 2:    record
 3:      Value : Integer;
 4:       _object : aliased GNARL.Protection_Entries
 5:                               (<Num_Entries>);
 6:    end record;
 7:
 8: procedure Finalize (_object : poV) is
 9: begin
10:    -- Raise Program_Error to the queued tasks.
11:    ...
12: end Finalize;
```

The protected type specification is expanded into a record type declaration (lines 1 to 6). If the protected type has discriminants, the record type has the same discriminants. The record type is defined as limited-controlled. Limited because, in analogy to task types, protected types are limited types [AAR95, Section 9.4(23)], and hence have neither an assignment operation nor predefined equality operators. It is also controlled to implement the clean-up action described above: when the object is finalized, each call remaining on any entry queue of the object must be removed from its queue and Program_Error must be raised at the place of the corresponding entry-call statement [AAR95, Section 9.4(20)]. PO's private data is expanded into the components of the record-type declaration (line 3). The field *_object* (line 4) contains additional run-time data: the lock, entry queues, the priority of the object, etc.

11.2.2 Expansion of Protected Subprograms

For each protected operation Op, the GNAT compiler generates two subprograms: OpP (the protected version) and OpN (the non-protected one). OpP simply takes the object lock, and then invokes OpN. OpN contains the (suitably expanded) user code. If a call is an internal call, i.e. a call from within an operation of the same object, the call invokes OpN directly. If the call is external, it is implemented as a call to OpP. In addition, one additional parameter is added by the compiler to the parameters profile of the protected subprograms: the _object. Because protected procedures can modify the object's state, they receive the object as **in out** mode parameter. Protected functions receive the object as an **in** mode parameter. For example:

```
procedure procP (_object : in out poV; ... );
procedure procN (_object : in out poV; ... );
```

A reference to a component of the object in the body of an operation, must be transformed into a reference to the component of the run-time object on which the operation is applied. This is implemented by introducing renaming declarations in the expanded subprograms. For example, the component Value in the definition above, leads to the following local declaration in all expanded subprograms:

```
  Value : Integer renames _object.Value;
```

The local variables introduced in this fashion are called Privals in the GNAT sources. References to private components are replaced by references to Privals thoughtout the bodies of protected operations. Let us see the expansion of subprogram *procP* in detail.

```
 1: procedure procP (_object : in out poV; ... ) is
 2:   procedure Clean is
 3:   begin
 4:      GNARL.Service_Entries (_object._object'access);
 5:      GNARL.Unlock (_object._object'access);
 6:      GNARL.Abort_Undefer;
 7:   end Clean;
 8: begin
 9:    GNARL.Abort_Defer;
10:    GNARL.Lock_Write (_object._object'access);
11:    begin
12:       procN (_object; ... );
13:    exception
14:       when others =>
15:          declare
16:             E : Exception_Occurrence;
17:          begin
18:             GNARL.Save_Occurrence
19:                (E, GNARL.Get_Current_Exception);
20:             Clean;
21:             GNARL.Reraise (E);
22:          end;
23:    at end
24:       Clean;
25:    end;
26: end procP;
```

Protected operations are abort-deferred operations [AAR95, Section 9.8(5-6)]. Therefore, the *P* subprogram calls the run-time to defer the abortion (line 9) and to obtain the read/write access of the PO (line 10), and finally calls the *N* subprogram (line 12). On return from *N* we have two possible scenarios: No exception was raised by the user code. In this case the at-end statement (line 24), an internal statement of the GNAT compiler which is executed whether the code was executed successfully or not, calls the local subprogram *Clean* to reevaluate the barriers and service queued entry-calls (line 4), unlock the protected object (line 5), and undefer the abortion (line 6). Otherwise, if the execution of the *N* subprogram propagates an exception, the barrier must be reevaluated and the queued entry-calls must be serviced before propagating back the exception to the calling-task. Therefore, the exception handler saves the exception occurrence (line 18), calls the local subprogram Clean, and finally re-raises the exception (line 21).

11.2.3 Expansion of Entry Barriers

Entry barriers are expanded into functions that return a boolean type. As for other operations, the expansion adds one parameter to designate the object itself. Barriers can access all components of the object, and therefore the expansion includes the same object renamings as other protected operations. The front-end builds an array of pointers to the barrier functions, and the run-time invokes them indirectly. The expansion of the barriers is as follows:

```
function EntryBarrier
   (Object        : Address;
    Entry_Index   : Protected_Entry_Index)
   return Boolean
is
   <Discriminant_Renamings>
   <Private_Object_Renamings>
begin
   return <Barrier_Expression>;
end EntryBarrier;
```

11.2.4 Expansion of Entry bodies

Similar to the barriers, the entry bodies are expanded into procedures with the same profile. They are expanded as follows:

```
procedure EntryName
    (Object       : Address;
     Parameters   : Address;
     Entry_Index  : Protected_Entry_Index)
is
   <Discriminant_Renamings and Private_Object_Renamings>
   type poVP is access poV;
   function To_PoVP is new Unchecked_Conversion (Address, PoVP);
   _object : PoVP := To_PoVP (Object);
begin
   << Entry body statements >>
   GNARL.Complete_Entry_Body (_object._object);
exception
   when others =>
      GNARL.Exceptional_Complete_Entry_Body
           (_object._object, GNARL.Get_GNAT_Exception);
end EntryName;
```

Similar to the *N* subprograms (cf. Section 11.2.2), the compiler adds some renamings to facilitate the access to the discriminants and private state (line 6). Because the procedure receives as parameter the address of the object (not the object itself), the front-end also generates the unchecked conversion of this address to the corresponding access to the object (lines 7 to 9). In addition, the front-end also generates calls to notify the run-time the successful execution of the entry-body statements (line 12) or unsuccessful execution (line 15).

11.2.5 Table to Barriers and Entry-Bodies

In addition to the barrier and entry-body expansion described above, the front-end also generates a table initialized with the access to the expanded subprograms. Each element has the access to an entry-barrier expanded function and the access to an entry-body expanded procedure. The front-end also generates a call to the run-time to pass this table, which is used by the run-time to evaluate the entry-barriers and to call the selected entry-body.

11.2.6 Expansion of Entry Families

For each entry-family the front-end adds one field to the PO type-declaration. This field saves the bounds of the entry-family specification. The element-type of these arrays is set to void because the contents of the array are not used. The protected type is then expanded as follows:

```
type poV (Discriminants) is new Limited_Controlled with
record
   <Private_Data>
   _object : aliased GNARL.Protection_Entries (<Num_Entries>);
   Entry_Family_Name : array (<Bounds>) of Void;
end record;
```

11.3 Optimizations

To realize simple and efficient synchronization regimes, a protected object without entries is sometimes sufficient. Such an object can be implemented more efficiently, and is recognized by the GNAT front-end. The resulting expansion is simpler, as indicated in the following example:

```
type poV (Discriminants) is limited record
   <private-data-fields>
   _object : aliased Protection;
end record;
```

Because now the protected object has no queues, the expanded type is simplified as follows 1) It is not a controlled type, because there is no need to raise Program_Error on queued entries, and 2) The _object component is smaller (*Protection* type only has the object lock and the PO ceiling).

In the absence of entries the run-time is also able to provide a faster implementation for protected objects. For this purpose, the GNAT run-time provides a second set of subprograms which are called by the expanded code in case of protected objects without entries (see package System.Tasking.Protected_Objects). In addition, the expander does not generate the call to re-evaluate the barriers after the execution of the body of a protected procedure.

11.4 Summary

There are two models to implement the protected objects: the self-service and the proxy model. GNAT follows the proxy model because the implementation of the self-service model with Pthreads is not feasible at a low cost. There are also two main implementations of the proxy model: the in-line and the call-back implementation. The main difference between both implementation resides on the service-entries routine. In the in-line model it is generated by the expander; in the call-back model, it is inside the run-time. GNAT follows the call-back implementation because the call-back interface allows for much simpler translations, and the expanded code is more understable.

Protected subprograms are translated to two subprograms: P and N. P obtains the object lock calls N, which has the user code. The barriers are expanded into functions that return a boolean data-type, and the entry-bodies are expanded into procedures. The front-end also generates a table with access to these subprograms. This table is used by the run-time to evaluate the barriers and call the selected entry-body.

Chapter 12

Expansion of Controlled-Types

Controlled-types[1] are tagged types that support automatic initialization and reclamation. As such, they provide capabilities analogous to constructors and destructors in C++. Automatic reclamation of complex objects with dynamically allocated components goes a long way to compensate for the absence of real garbage collection in Ada95. During the design phase of the language, it was proposed that all tagged types should provide this capability. For various technical reasons, this ambitious proposal was abandoned, and controlled types where placed in a special category: all controlled types derive from a predefined tagged type, and as such all of them inherit three operations: *Initialize*, *Adjust*, and *Finalize* [AAR95, Section 7.6]. The predefined library package *Ada.Finalization* declares the root type *Controlled* and its three primitive operations. The package also includes *Limited_Controlled*, a type whose descendants are all limited, and which only have the primitive operations Initialize and Finalize. The language semantics specifies that Initialize is automatically invoked upon the declaration of an object of a controlled type, if the declaration does not include an explicit initialization; Finalize is automatically invoked when the object is about to go out of scope, i.e. when the scope that declares it is about to be completed. Finalize performs whatever clean-ups are desired (for example, deallocation of indirect structures, release of locks, closing of files, etc.). Finally, Adjust is called on the left-hand-side of an assignment-statement *Obj1 := Obj2*, after Obj1 is finalized, and the new value Obj2 has been copied into Obj1. Adjust is not defined for limited controlled types. The invocation of these operations is an important part of the expansion phase of the compiler.

When a scope contains several controlled-type objects, each object is initial-

[1]The contents of this chapter are based on the paper [CDG95].

ized in the order of its declaration within the scope. Upon scope exit the objects are finalized in the reverse order. This reverse order is important, since later objects may contain references to earlier objects. If an exception occurs during initialization, then only those controlled objects that have been initialized so far will be finalized.

The primitive operations of controlled types apply not only to stand-alone declared objects, but also to dynamically-allocated objects and controlled components of composite objects. A dynamically allocated object is finalized either when the scope of its associated access-type is exited, or when the programmer explicitly deallocates it. In the case of controlled components of a composite object, the controlled components are finalized in the reverse order of their initialization within the containing object. In addition, Adjust is called when the controlled components are either assigned individually, or upon assignment to their enclosing object. If a controlled object includes controlled components, Initialize or Adjust is first invoked on the components and then on the enclosing object; Finalize is called in the reverse order. Finalization actions also occur for anonymous objects such as aggregates and function results. For these special objects, the finalization will occur upon completion of execution of the smallest enclosing construct, once the value of the aggregate or function result is no longer needed.

12.1 Implementation Overview

The rules described above immediately suggest that the run-time must include some data structure that groups all controlled objects in a given scope. This structure must be dynamic (i.e. a list) because the front-end cannot ascertain statically how many controlled objects will be created in a given scope. Therefore, for each scope that contains declarations of controlled objects (or objects with controlled components) the front-end creates a local variable that holds the head of the list. The front-end then generates code that attaches each controlled object to this list. Finally the front-end generates clean-up code for each such scope. The clean-up code traverses the list in order to perform the appropriate finalization operations on each object. This mechanism requires that controlled objects have some compiler-generated component that holds a pointer to next object in the list (this is in addition to the tag component that is generated for all objects of a tagged type). In fact, finalization lists are doubly-linked, because deletions must at times be performed on them. The two link components are inherited from the root type of all controlled types.

The finalization of objects with controlled components requires additional machinery within the object itself. Consider a variant record, some of whose components may be controlled. It is necessary to locate those components from the enclosing object itself to finalize them when the enclosing object is finalized. As a result, such records include an additional component, called *Controller*, that anchors the local list of controlled components. The finalization code must traverse this local list as well.

Not surprisingly, there are some complex interactions between finalization and other languages features. In the following sections we discuss the most interesting of these issues, and subsequently discuss their solution in GNAT.

12.1.1 Exceptional Block Exit

The initialization/finalization mechanism must be robust in the presence of exceptions: one of the purposes of finalization is to avoid any memory leaks from the creation of local objects. This purpose must be realized as well in the presence of an abnormal scope termination, such as when an exception is raised, or when the task containing the block is aborted, there may exist objects which have not yet been created and received proper initialization. For this objects, Finalize must not be called. For instance in the folowing code:

```
declare
   S1 : Some_Controlled_Type;
   X  : Pos := Random (0,1);    -- Constraint_Error is
                                -- randomly raised.
   S2 ; Some_Controlled_Type;
begin
   null;
end;
```

S1 is initialized, but S2 might not be. Consequently finalization should always occur for S1 whereas S2 should be finalized only if it has been initialized. Thus an implementation which expands calls to *Finalize* at the end of the block is inadequate. As a further complication, note that exceptions may be raised during initialization of composite object containing controlled components, in which case only the initialized components of the object needs finalization.

12.1.2 Finalization of Anonymous Objects

Finalization actions for anonymous objects must occur upon completion of execution of the smallest enclosing construct, that is, as soon as their value is no longer needed. Again, this mechanism has to work even if an exception is raised in the middle of executing the construct. The following code present two examples. *Empty* is a function returning a controlled object, and *Exists* is a function that takes such an object as a parameter. The call to Empty creates an anonymous object that must be finalized when the enclosing call to *Exists* returns:

```
declare
   X : Boolean := Exists (1, Empty);
   -- The result of the call to Empty is kept in an
   -- anonymous object during the execution of Exists,
   -- and Finalize must be invoked no latter than
   -- the semicolon.
begin
   if Exists (2, Empty) then
      ...
   else
      ...
   end if;
   -- Here the anonymous object has to be finalized before
   -- the execution of either branch of the if statement.
end;
```

12.1.3 Finalization of Dynamically Allocated Objects

Recall that dynamically objects can be reclaimed when the corresponding access type goes out of scope. This rule extends to controlled objects: if an object is allocated dynamically, it must be finalized when the the scope of the access type is completed. Of course, if a dynamic object is deallocated explicitly, it must be finalized before the final storage reclamation. The implementation must then attach dynamically allocated objects to the finalization list of the scope of the type itself. The expanded code for any use of Unchecked_Deallocation must include an invocation of Finalize when the designated object is controlled.

12.1.4 Problems Related to Mutable Objects

A variable of a discriminated type with defaulted distriminants may contain differing numbers of controlled components at different times. This possibility introduces an asymmetry between elaboration and finalization. In the following example no controlled components are present at the beginning of the execution, but after the assignment, X will contain three such components:

```
declare
   type T_Table is array (Natural range <>)
                          of Some_Controlled_Type;
   subtype Index is Natural range 0 .. 10;
   type Rec (N : Index := 0) is record
        T : T_Table (1 .. N);
   end record;
   X : Rec;
begin
   X := (3, (1 .. 3 => Empty)); -- 3 Controlled components.
end;
```

This example makes it clear that objects with controlled components must include some additional data-structures to keep track of their changeable controllable contents. In addition, such objects are not necessarily controlled themselves, so, the chaining mechanism must include some level of indirection to denote these objects, given that they do not have the link field of controlled objects. Arrays of controlled objects are yet another complication, because there is nowhere to place additional pointers to link the components.

12.1.5 Controlled Class-Wide Objects

Type extensions can introduce additional controlled components. Given that a class-wide object can denote any descendant of a given type, we must assume that in general it may include controlled components, even if the ancestor type does not. This forces the compiler to make a worst-case assumption for class-wide objects and parameters. Consider the following case:

```
package Test is
   type T is tagged null record;
   function F returns T'Class;
end Test;

with Test; use Test;
procedure Try is
   V : T'Class := F;
   -- Does F yield a value containing controlled components?
begin
   ...
end Try;
```

The expanded code must assume that *Try* is a scope that needs finalization. Therefore it must create a finalization list, and generate code to attach the anonymous object returned by the call to F to this list, in some indirect fashion because the object might not be controlled after all.

12.2 Expansion Activities for Controlled-Types

For each block that contains objects, the expander generates a *Finalization Chain*. When a controlled object is elaborated, it is first *Initialized* or *Adjusted* (depending on whether an initial value was present or not), then attached at the beginning of this chain. For example, let us assume the following declarations:

```
declare
   X : Some_Controlled_Type;
   Y : Some_Controlled_Type := X;
begin
   << Additional user code >>
end;
```

This fragment is expanded as follows:

```
declare
   F : GNARL.Finalizable_Ptr;
begin
   X : Some_Controlled_Type;
    Initialize (X);
   GNARL.Attach_To_Final_List (F, Finalizable (X));
   Y : Some_Controlled_Type := X;
```

```
      Adjust (Y);
      GNARL.Attach_To_Final_List (F, Finalizable (Y));

      << Additional user code >>
   at end
      GNARL.Finalize_List (F);
   end;
```

Finalizable_Ptr is an access to the class representing all controlled objects. Since objects are inserted at the beginning of the list, the ordering of the chain is exactly correct for the required sequence of finalization actions. The fact that the chain is built dynamically ensures that only successfully elaborated objects are dealt with in case of exceptional exit. Upon scope exit, the at-end statement ensures that Finalize_List is called whether the scope was successfully executed or not. Finalize_List is a run-time subprogram that finalizes all objects on the list, by dispatching to the Finalize procedure of each. The list is of course heterogeneous because Finalize_Ptr is an access-to-class-wide type, and any object whose type is derived from Controlled can be attached to this list.

12.2.1 Expansion of Assignment

At a first sight, the expansion of the the assignment statement *Obj1 := Obj2* might be:

```
      Finalize (Obj1);    -- discard old value
      Obj1 := Obj2;
      Adjust (Obj1);      -- remove accidental sharing on new value
```

However, various problems make such an implementation unworkable. First, Obj1 may refer to objects present in Obj2 and thus cannot be finalized before Obj2 is evaluated. Second, the assignment itself must be specialized since copying the hidden pointers that attach objects to finalization lists is clearly nonsensical. Third, the self-assignment (*X := X*), although not a particularly useful construct, does not work, because it would finalize the target of the assignment as well. This case must be addressed specially, either by introducing a temporary object or by avoiding the execution of any finalization actions. Thus, the front-end expands assignment as follows, which works in the general case and can be often be optimized:

```
Anon1 : Some_Controlled_Type renames Obj1;
Anon2 : Address := Obj2'Address;

if Anon1'Address /= Anon2 then -- Protect against X := X
   Finalize (Anon1);
   GNARL.Copy_Explicit_Part (Anon2.all, to => Anon1);
   Adjust (Anon1);
end if;
```

Note that the target object, even though it has been finalized, remains in the finalization list because it still need to be finalized upon scope exit. In general finalization is idempotent, i.e. finalizing an object twice is a no-op.

12.2.2 Expansion of Anonymous Controlled Objects

Some constructs such as aggregates and functions generate anonymous objects that are part of some enclosing expression or construct. When such objects are controlled, they must be finalized as soon as they are no longer needed, that is to say before the beginning of the next statement. The GNAT expander generates *transient blocks* to handle anonymous objects; these blocks are placed around the construct that uses the intermediate objects. Such blocks will contain the declaration of a finalization list, and will cause the generation of finalization code as for user-declared constructs. Consider the previous example: function *Empty* yields a controlled value that is only used during the execution of *Exists*:

```
  X := Exists (1, Inside => Empty);
```

GNAT expands this code as follows:

```
declare
  Anon : Some_Controlled_Type := Empty;
begin
  X := Exists (1, Inside => Anon);
end;
```

An intermediate block can be introduced without changing the semantics of the program in order to make the anonymous object and the corresponding finalization list explicit. This new block contains a controlled object and thus will be expanded using the scheme discussed above (cf. Section 12.2). The same mechanism can be extended to deal with anonymous objects that appear in flow-of-control structures (such as if and while statements).

The problem is a bit more complex when controlled anonymous objects appear in a declaration, since blocks are not allowed in such a context. To handle this case, the anonymous object is attached to an intermediate finalization list which is finalized right after the declaration. For example:

```
declare
   B : Boolean := Exists (1, Empty);
begin
   ...
end;
```

It is expanded into:

```
declare
   Aux_L  : GNARL.List_Controller;
   Anon   : Some_Controlled_Type;
   B      : Boolean;
begin
   Anon := Empty;
   Adjust (Anon);
   GNARL.Attach_To_Final_List (Aux_L, Anon);
   B := Exists (1, Anon);
   GNARL.Finalize (Aux_L); --<<-- Finalize here, not at the
                           --      end of the block
   ...
end;
```

List_Controller is itself a controlled type. Thus, an object of that type is attached to the enclosing scope's finalization chain, ensuring that the anonymous object will be finalized even if an exception is raised between its definition and the finalize call. In the normal case, the List_Controller is finalized twice, once right after the declaration, and once again upon scope exit. Therefore the run-time Finalize routine makes sure that the second finalization has no effect.

12.2.3 Objects with Controlled Components

Composite type such as records and arrays can contain controlled components, and the expander must take care of calling the proper Initialize, Adjust and Finalize routines on their components. For this purpose the expander generates implicit procedures called *_Deep_Initialize*, *_Deep_Adjust* and *_Deep_Finalize* that are used in a manner similar to their counterparts for regular controlled types. These procedures are specialized according as to whether they handle records or arrays. Here is the body of _Deep_Adjust for a type T that is a one-dimensional array of controlled objects:

```
procedure _Deep_Adjust (V : in out T;
                        C : Final_List;
                        B : Boolean) is
begin
   for J in V'range loop
       _Deep_Adjust (V(J));
       if B then
          Attach_To_Final_List (C, V (J));
       end if;
   end loop;
end _Deep_Adjust;
```

Note that the deep procedures have a conditional mechanism to attach objects to the finalization chain so that the same procedure can be used in a context where attachment is required, such as explicit initialization, as well as when it is not needed, such as in the assignment case. Note also the recursive nature of the above definition: Deep_Adjust on an array is defined in term of Deep_Adjust of its components. Ultimately, if the component type is a simple controlled type with no controlled components Deep_Adjust ends up just begin a call to the user-defined Adjust subprogram.

A similar approach could have been used for records. In that case, deep procedures would have been implicitly defined to perform the finalization actions on the controlled components in the right order, depending of the structure of the type itself. The controlled components would have been stored on the finalization list of the enclosing scope. Unfortunately such a model makes the assignment of mutable objects quite difficult: the number of objects on the finalization list may be different before and after the assignment, so all the controlled components of the target would need to be removed from it before the assignment and afterwards put back at the same place on the list. To avoid such a problem as well as to simplify the definition of deep procedures for records a different approach has been used. Records are considered as scopes and they have their own internal finalization chain on which all controlled components are chained. This is achieved by inserting a hidden Record_Controller component within the record itself. For example:

```
type Rec (N : Index := 0) is record
   _Controller : GNARL.Record_Controller;
   T : Sets (1 .. N);
end record;
```

Record_Controller plays a role equivalent to List_Controller: it introduces an indirection in the enclosing finalization list. The finalization list of controlled components is local to the object. So, upon assignment the number of controlled components may vary without affecting the enclosing finalization list. This provides a simple solution to the mutable record problem.

Class-wide objects present a interesting challenge since the compiler doesn't know how many, if any, controlled components are present in such objects. To address this problem, class-wide types are considered "potentially" controlled and calls to the *deep* procedures are always generated for initializations and assignments. Dispatching is used to ensure that the appropriate *deep* procedure is called.

12.3 Summary

Controlled-types introduce interesting implementation problems that impose a close cooperation between compile-time and run-time activities. The expander generates declarations for data structures at the scope level and at the type level, that are used to create run-time lists that hold controlled objects. The expander also generates calls to initialization and finalization routines. The run-time includes general purpose finalization routines that traverse these lists and dispatch to type-specific finalization operations. The expander adds blocks around the use of anonymous controlled objects, to ensure that they are finalized in a timely fashion. The resulting mechanism handles dynamically allocated objects and objects with controlled components as well.

Chapter 13

Expansion of Tagged-Types

Tagged-types give linguistic support to the two basic object-oriented concepts: type-extension and inheritance. In Ada95, a tagged-type defines a class that designates a set of types derived from a common root, thus sharing properties and enabling *class-wide programming*. Note that the word **class** has a slightly different meaning in most other object-oriented languages, where it is used to designate a single type and not a hierarchy of types.

It is important to distinguish clearly between *primitive operations* of a tagged type, which have one or more parameters of their associated type (or an anonymous access to it), and subprograms operating on class-wide objects (or access to class-wide objects). The former are inherited when the type is extended, and may be redefined in an extension, each version applying to its specific type only. By contrast, subprograms with class-wide formals have a single definition and apply to all members of the class. Dynamic dispatching occurs whenever a primitive operation is called and (at least) one of its formal parameters has a specific tagged-type and its corresponding actual is class-wide; this parameter is called a *controlling argument*. The tag of the actual determines which version of the primitive operation is to be called.

Most of the implementation of tagged types in GNAT follow the ideas discussed in [Dis92] and [CP94]. This chapter presents the main aspects of this model.

13.1 Tagged and Polymorphic Objects

The intuitive idea behind tagged-types is that values of such a type carry a *tag* which is used, among other things, to relate the run-time value to its original type and to the operations that apply to it. This tag allows the simple implementation of run-time type-specific actions, such as dynamic dispatching and membership testing. Let us consider the following declaration of a root tagged-type:

```
type My_Tagged is abstract tagged
   record
      ...
   end record;
```

It is transformed by the GNAT front-end by the addition of a new component, whose predefined type appears in Ada.Tags:

```
type My_Tagged is abstract tagged
   record
      _Tag : Tags.Tag;
      ...
   end record;
```

Each tagged-type has its own tag. A copy of the tag is inserted into every object of the type. The value of the tag is a pointer to a type-specific data area, that includes first and foremost a table of primitive operations of the tagged-type. This table is built at the point the type is frozen. The tag of an object is constant, and cannot be modified after the object is created. (Note that a conversion cannot affect the tag of an object, its purpose is to allow the object to be viewed as another member of the class - a view conversion - but does not actually change the value of the tag stored in the object).

By definition a type extension inherits all the components of its parent. We could describe the type extension by copying explicitly all the component declarations of the parent and appending the components in the extension. It turns out to be simpler to describe all the inherited components as being part of a single inherited collective component, called _Parent. The following type extension:

```
type My_Ext_Tagged is new My_Tagged with
   record
      ...
   end record;
```

It is expanded as follows:

```
type My_Ext_Tagged is
   record
      _Tag    : Tags.Tag;
      _Parent : My_Tagged;
      ...
   end record;
```

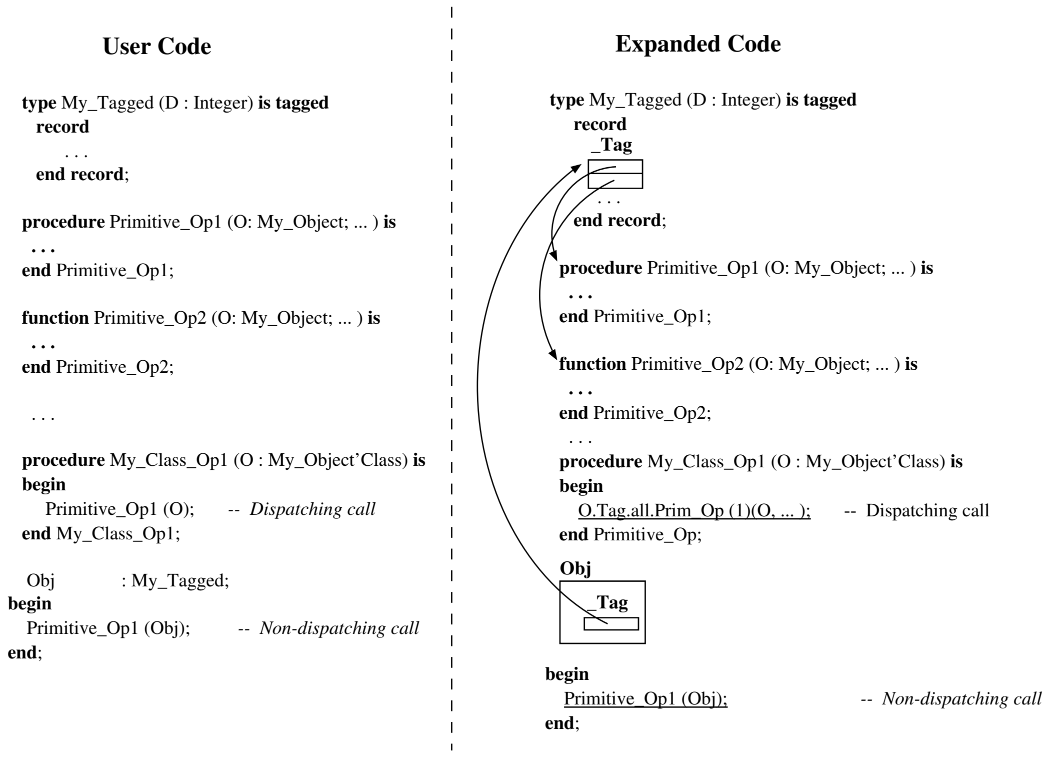
The main advantage of this technique is that it transforms record extensions into regular records.

For many reasons, it turns out to be mandatory that any field in a tagged type keep the same location within the record in any descendant type. To start with, dynamic dispatching depends on the value of the tag, and thus the tag must appear at the same location in all tagged types. Additionally, for a class-wide object whose actual type is not known until run-time, selecting any component would be impossible (or prohibitively inefficient) if the position of the field depended on the type. By having all the inherited components appear conceptually as part of a single _Parent component type insures that the layout of the parent is respected. This component is inserted at the beginning of the record. Components of the extension follow.

13.2 The Dispatch Table

A dispatching call consists in selecting and calling the version of a primitive operation that applies to the type of its controlling argument. Recall that in Ada95, unlike C++, whether a call is dispatching or not depends on whether there is a class-wide actual. If the actuals are statically tagged, the compiler can determine the operation to be called. If an actual is dynamically tagged, the run-time call must be indirect: the proper operation must be selected from the **dispatch table** of the type of the actual. The dispatch table is a table of subprogram addresses, and a dispatching call is an indirect call through an entry in the table. The position of each primitive operation in the table is established by the compiler, so the call uses a static offset to determine the address of the subprogram to be called. (cf. Figure 13.1).

The left side of figure 13.1 shows the declaration of a tagged-type, some user-defined primitive subprograms for it (My_Primitive_OpN), one non-primitive dispatching subprogram (My_Class_Op1), one object declaration of the type (Obj),



```
type My_Tagged (D : Integer) is tagged
  record
     . . .
  end record;

procedure Primitive_Op1 (O: My_Object; ... ) is
 . . .
end Primitive_Op1;

function Primitive_Op2 (O: My_Object; ... ) is
 . . .
end Primitive_Op2;

 . . .

procedure My_Class_Op1 (O : My_Object'Class) is
begin
   Primitive_Op1 (O);      -- Dispatching call
end My_Class_Op1;

  Obj          : My_Tagged;
begin
  Primitive_Op1 (Obj);        -- Non-dispatching call
end;
```

```
type My_Tagged (D : Integer) is tagged
  record
    _Tag
     . . .
  end record;

procedure Primitive_Op1 (O: My_Object; ... ) is
 . . .
end Primitive_Op1;

function Primitive_Op2 (O: My_Object; ... ) is
 . . .
end Primitive_Op2;
 . . .
procedure My_Class_Op1 (O : My_Object'Class) is
begin
  O.Tag.all.Prim_Op (1)(O, ... );     -- Dispatching call
end Primitive_Op;

Obj
  _Tag

begin
  Primitive_Op1 (Obj);                   -- Non-dispatching call
end;
```

Figure 13.1: Dispatch Table Example.

and a non-dispatching call to one primitive operations. On the right side we display the expanded code produced by the GNAT front-end. The tagged-type generates a data structure that is essentially an array of addresses, of which the dispatch table is the central component (we discuss later the other components of the **type-specific data**). The front-end also generates code to initialize the dispatch table, by storing in it the addresses of the primitive subprograms of the type. In an object, the tag component is a pointer to the dispatch table.

In addition to dispatching calls, a predefined operation that requires run-time type information is the membership test: *X in T*. When *T* is a specific tagged-type, this test consists simply of verifying that X's tag points to the dispatch table for the type T (for convenience, the tag of T is also present in the type-specific data for T). When the test is of the form *X in T'Class*, the problem is more complex, because the tag of X can be a pointer to any descendant of T. Two implementations are possible for this test. One can store in the type-specific data a pointer to the dispatch table of the immediate ancestor. In this case the membership test consists of traversing the list of ancestors' dispatch tables, and return True if the dispatch table for *T* is found during the traversal. The other implementation stores the tags of all the ancestors in the type-specific data, along with the inheritance depth (the total number of ancestors of the type, i.e. the distance to the root in the type hierarchy). The difference in depths between the type of *X* and *T* gives the actual location where *T* must be found in the table of ancestor tags for the membership to succeed. GNAT implements this later approach because it ensures that the evaluation of the membership test takes constant time (see details in 13.3).

13.3 Primitive Operations

The following subprograms are present for all tagged types, and provide the results of the corresponding attributes. The bodies of these subprograms are generated by the front-end, at the point at which the type is frozen. These subprograms must in general be dispatching, since they can apply to class-wide objects, and the value produced will vary with the actual object subtype: _Alignment, _Size, _Read, _Write, _Input, and _Output. In addition, the following subprograms are present for non-limited tagged-types, and implement additional dispatching operations for predefined operations: _equality, _assign, _deep_finalize, and _deep_adjust. The latter two are empty procedures, unless the type contains some controlled components that require finalization actions (the deep in the name refers to the fact that the action applies recursively to controlled components, cf. Chapter 12).

For example, let us assume the following tagged-type declaration:

```
type My_Record is tagged
   record
      My_Data : Integer;
   end record;
```

The GNAT front-end generates the equivalent of the following expanded code from it:

```
-- Declaration of the type-specific data
type My_Record_Data is record
   DT      : array (1 .. 11) of Address;      -- Dispatch table
   TSD     : array (1 .. 3)  of address;      -- Ancestor info
   _Parent : Tags.Tag := DT'Address;          -- Tag of the type
   _F      : Boolean := True;
   _E      : constant String := "My_Object"; -- Debug info
end record;
```

```
function _Alignment
  (X : My_record) return Integer;
function _Size
  (S : My_Record) return Integer;
procedure _Read
  (S : Root_Stream_Type; V : out My_Record);
procedure _Write
  (S : Root_Stream_Type; V : My_Record);
function _Input
  (S : access Root_Stream_Type) return My_Record;
procedure _Output
  (S : access Root_Stream_Type; V : My_Record);
function "="
  (X : My_Record; Y : My_Record) return  Boolean;
procedure _Assign
  (X : out My_Record; Y : My_Record);
procedure _Deep_Adjust
  (L : in out Finalizable_ptr;
   V : in out My_Record;
   B : Integer);
procedure _Deep_Finalize
  (V : in out My_Record; B : Boolean);
...
-- The initialization code for the dispatch table includes
-- the following for all primitive operations

   My_Record_Data.DT (1) := _Alignment'Address;
   My_Record_Data.DT (2) := _Size'Address;
...
```

The format of GNAT's dispatch table is customizable in order to match the format used by other object-oriented languages. GNAT supports programs that use two different dispatch table formats at the same time: the native format, that supports Ada 95 tagged types and which is described in Ada.Tags, and a foreign format for types that are imported from some other language (typically C++) which is described in interfaces.cpp. Several pragmas are provided to allow the user to specify the position of the tag in the foreign layout. The boolean field (_F) is used for elaboration purposes. Finally the name of the tagged-type is expanded in a string-type field (_E). This field is used by the debugger.

After the record type, the expander creates the bodies of the default primitive operations. By default, the bodies of _Read, _Write, _Deep_Adjust, and _Deep_Finalize are empty. The other subprograms are expanded as follows:

```
function _Alignment (X : My_Record) return Integer is
begin
   return X'Alignment;
end _Alignment;

function _Size (X : My_Record) return Integer is
begin
   return X'Size;
end _Size;

function _Input (S : Root_Stream_Type) return My_Record is
   V : My_Record;
begin
   _Read (S, V);
   return V;
end _Input;

procedure _Output (S : access Root_Stream_Type;
                   V : My_Record) is
begin
   _Write (S, V);
end _Output;

function "=" (X : My_Record; Y : My_Record) return Boolean is
begin
   return X.My_Data = X.My_Data;
end "=";

procedure _Assign (X : out My_Record; Y : My_Record) is
begin
   if not (X'Address = Y'Address) then
      declare
        Aux : Tags.Tag := X.Tag;
      begin
        X := Y;
        X.Tag := Aux;
      end;
   end if;
exception
   when others =>
     GNARL.Undefer.all;
     raise Program_Error;
end _Assign;
```

_Alignment and _Size return the value of the corresponding attribute applied t to the object; _Input and _Ouput call the corresponding _Read and _Write primitives (both being null by default, this does nothing); "=" is expanded into the required code to compare all the user-defined components of the tagged-type. Finally, the body of the primitive operation _Assign would seem to be just *X:=Y*. However, the expanded code protects the user from self assignment (*X:=X*), which is incorrect if the object is controlled (the finalization of the old value of the left-hand side would end up destroying the object itself). Finally, in order to handle correctly an assignment whose right-hand side is a conversion, the assignment must first preserve the tag of the target, perform the assignment, and finally reset the tag. Let us examine the following example:

```
declare
   type My_Record is tagged
      record
         Some_Data : Integer;
      end record;

   type My_Extension is My_Record with
      record
         More_Data : Character;
      end record;

   Obj1 : My_Record;
   Obj2 : My_Extension;
begin
   Obj1 := My_Record (Obj2); -- Explicit conversion
end;
```

After the elaboration of the two objects, the tag of Obj1 points to the dispatching table of My_Record and the tag of Obj2 points to the dispatching table of My_Extension. The conversion does not change the tag of Obj2, but simply indicates that Obj2 is to be regarded as having the ancestor type in the context of the assignment. If the compiler were to implement the assignment as a copy of the whole contents, the tag of Obj1 would point to the dispatching table of My_Extension which would be clearly incorrect.

13.4 Summary

Objects of a tagged-type include a *tag* which is used at run-time to implement object-oriented notions of polymorphism and dynamic dispatching. The GNAT expander translates tagged-types into record types with a dispatch table; the tag in an object is a pointer to the dispatch table of the type. Objects of a type extension include a collective component that corresponds to all the components inherited from the parent. The GNAT front-end generates a number of dispatching primitive operations for several type attributes, and generates code to initialize the dispatch table by placing in it the addresses of all primitive operations.

Part IV

<u>Fourth Part: Run-Time</u>

Chapter 14

Tasking

Ada tasks semantics requires run-time support for storage allocation, task scheduling, and intertask communication. These functions are typically performed by the kernel of the operating system. Ada is so specific in its semantic requirements, however, that it is unlikely that a given existing operating system will make such services available in a form that can be directly used by the generated code. As a consequence, the compiler run-time must add routines that support Ada tasking semantics on top of OS primitives, or else provide a tasking kernel for applications that run on bare boards.

The GNAT run-time assumes that functionality equivalent to that of the POSIX threads library (pthreads) is available in the target system. The additional run-time information concerning each Ada task (task state, parent, activator, etc. [AAR95, Chapter 9]) is stored in a per-task record called the *Ada Task Control Block (ATCB)*.

14.1 The Ada Task Control Block

The Ada Task Control Block (ATCB) is a discriminated record, whose discriminant is the number of task entries (a central component of the ATCB is the array of queues whose size is fixed by the discriminant). In order to support Ada95 task discriminants and some Ada task attributes, the front-end generates an additional record (described in detail in Section 9.4). When a task is created, the run-time generates a new ATCB and links the new ATCB with its corresponding high-level record and *Threads Control Block (TCB)* record in the POSIX level (cf. Figure 14.1). In addition, the GNAT run-time inserts the new ATCB in a linked list (*All Tasks List*). ATCBs are always inserted in LIFO order (as a stack). Therefore, the first ATCB in this list corresponds to the most recently created task.

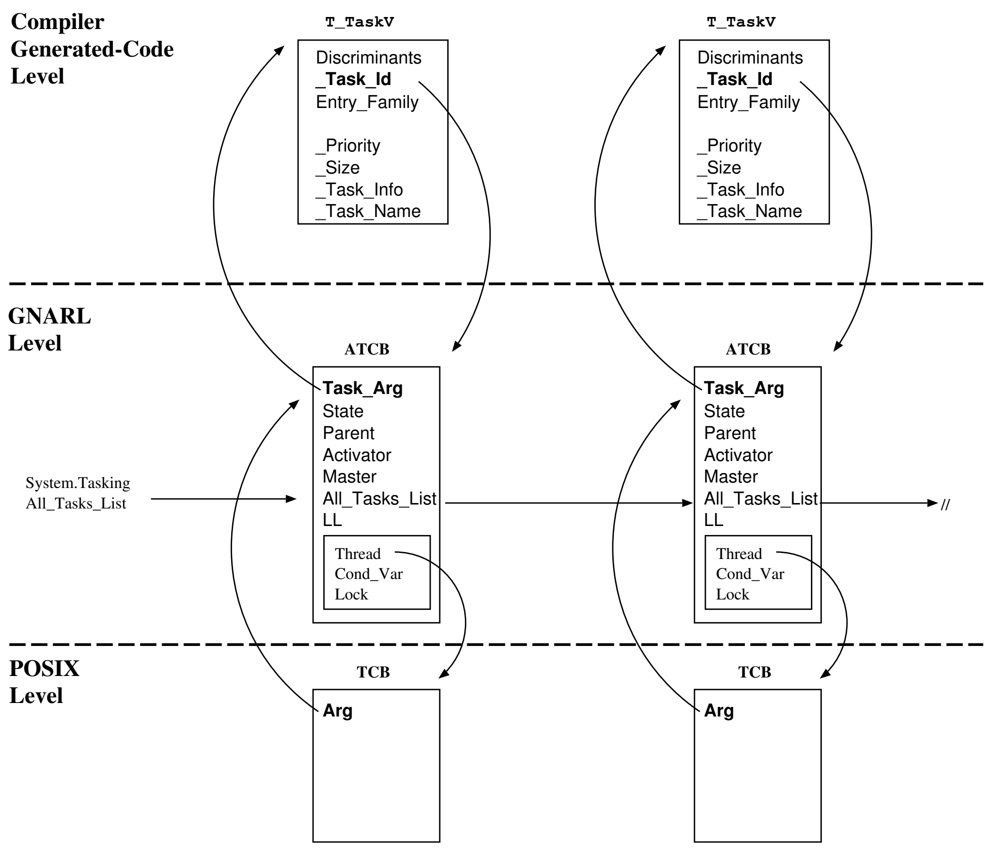


Figure 14.1: Run-Time Information Associated with Each Task.

14.2 Task States

GNAT considers four basic states over the lifetime of a task. The current state is indicated by the *State* ATCB field):

- *Unactivated*. The ATCB has been created and inserted in the *All Tasks List*, but no thread of control has been assigned to execute its body.
- *Runnable*. The task is executing (although it may be waiting for a mutex).
- *Sleep*. The task is blocked. The execution of a task may be blocked when it is forced to wait for some event external to the task. Examples of such

events are self-imposed time delays, termination of a subordinate task, and completion of the operations involved in intertask communication. A task that is not blocked is said to be ready.

- *Terminated*: The task is terminated, in the sense of ARM 9.3 (5). Any dependents that were waiting on Ada **terminate** alternatives have been awakened and have terminated themselves.

The sleep state is composed of the following sub-states:

- *Activator_Sleep*: Waiting for created tasks to complete activation.
- *Acceptor_Sleep*: Waiting on an accept or selective wait statement.
- *Entry_Caller_Sleep*: Waiting on an entry call.
- *Async_Select_Sleep*: Waiting to start the abortable part of an asynchronous select statement.
- *Delay_Sleep*: Waiting on a select statement with only a delay alternative open.
- *Master_Completion_Sleep*: Master completion has two phases. In the first phase the task, having completed a master within itself, waits for the tasks dependent on that master to become terminated or wait on a terminate phase.
- *Master_Phase_2_Sleep*: In phase 2 the task sleeps in Complete_Master, waiting for tasks on terminate alternatives to finish terminating.

14.3 Task Creation and Termination

According to Ada semantics, all tasks created by the elaboration of object declarations of a single declarative region (including subcomponents of declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together [AAR95, Section 9.2(2)]. In addition, if an exception is raised in the elaboration of a declarative part, then any task T created during that elaboration becomes terminated and is never activated. As T itself cannot handle the exception, the language requires the parent (creator) task to deal with the situation: the predefined exception *Tasking_Error* is raised in the activating context.

In order to achieve this semantics, the GNAT run-time uses an auxiliary list: the *Activation List*. The front-end expands the object declaration by introducing a local variable in the current scope, that holds the activation list (cf. Sections 9.1 and 9.6), and splits the OS call to create the new thread into two separate calls to the GNAT run-time: (1) Create_Task, which creates and initializes the new ACTB, and inserts it into both the all-tasks and activation lists, and (2) Activate_Task, which traverses the activation list and activates the new threads.

With respect to task termination, the concept of a *master* [AAR95, Section 9.3] is fundamental to the semantics of the language. (cf. Section 9.2). Basically a master defines a scope at the end of which the run-time must wait for termination of dependent tasks, before finalization of other objects created on such a scope. To implement this behavior, the front-end generates calls to the run-time subprograms Enter_Master and Complete_Master at the beginning and termination of a master scope (or, in the case of tasks, via Create_Task and Complete_Task subprograms). The GNAT run-time associates one identifier to each master, and two master identifiers with each task: the master of its Parent (*Master_Of_Task*) and its internal master nesting level (*Master_Within*). The identification method of masters provides an ordering on them, so that a master that depends on another will always have associated an identifier higher than that of its own master.

Normally a task starts out with internal master nesting level one larger than external master nesting level. This value is incremented by Enter_Master, which is called if the task itself has dependent tasks. It is set to 1 for the environment task. The environment task is the operating system thread which initializes the run-time and executes the main Ada subprogram. Before calling the main procedure of the Ada program, the environment task elaborates all library units needed by the Ada main program. This elaboration causes library-level tasks to be created and activated before the main procedure starts execution. Master level 2 is reserved for server tasks of the run-time system (the so called ”independent tasks”), and the level 3 is for the library level tasks.

- *Master_Of_Task* is set to 1 for the environment task. The level 2 is reserved for server tasks of the run-time (the so called *Independent Tasks*), and the level 3 is for the library level tasks. When a task is created it inherits the internal master nesting level of its Parent (the initial value of its *Master_Of_Task* is initialized with the current value of its Parent *Master_Within*). This value remains unmodified during the new task life and is used to ensure the Ada semantics for tasks finalization.

```
New_Task.Master_Of_Task := Activator.Master_Within
```

- *Master_Within* is set to the initial *Master_Of_Task* value plus one. When the tasks enters a scope with dependent tasks, its internal nesting level is incremented to one.

Tasks created by an allocator do not necessarily depend on their activator, but rather on the master that created the access type; in such case the activator's termination may precede the termination of the newly created task [AAR95, Section 9.2(5a)]. Therefore, the master of a task created by the evaluation of an allocator is the declarative region which contains the access type definition. Tasks declared in library-level packages have the main program as their master. That is, the main program cannot terminate until all library-level tasks have terminated [BW98, Chapter 4.3.2]. Figure 14.2 summarizes the basic concepts used by the run-time for handling Ada task termination.

Parent	The task executing the master on which T depends.
Activator	The task that created T's ATCB and activated it.
Master of Task	Parent's scope on which T depends.
Master Within	Nesting level of T's dependent tasks.

Figure 14.2: Definition of Parent, Activator, Master of Task and Master Within.

Example

In order to understand these concepts better, let's apply them to the following example:

```
procedure P is
   -- P:   Parent = Environment Task;
   --      Activator = Environment
   --      Master_Of_Task = 1; Master_Within = 2;

   task T1;
   -- T1: Parent = P; Activator = P
   --      Master_Of_Task = 2; Master_Within = 3;
   task body T1 is

      task type TT;
      task body TT is
      begin
         null;
      end TT;

      type TTA is access TT;
      T2 : TT;
      -- T2: Parent = T1; Activator = T1
      --      Master_Of_Task = 3; Master_Within = 4;

      task T3;
      -- T3: Parent = T1; Activator = T1
      --      Master_Of_Task = 3; Master_Within = 4;

      task body T3 is
         task T4;
         -- T4: Parent = T3; Activator = T3
         --      Master_Of_Task = 4; Master_Within = 5;
         task body T4 is
         begin
            null;
         end T4;
         T5 : TT;
         -- T5: Parent = T3; Activator = T3
         --      Master_Of_Task = 4; Master_Within = 5;
         T6 : TTA := new TT;
         -- T6: Parent = T1; Activator = T3
         --      Master_Of_Task = 2; Master_Within = 3;
      begin
         null;
      end T3;
   begin
     null;
   end T1;
begin
   null;
end P;
```

Parent and activator do not coincide for T6 because the task is created by means of an allocator, and in this case the parent of the new task is the task where the access type is declared, while the activator is the task that executes the allocator. In all other cases above, parent and activator coincide.

The abort statement is intended for use in response to those error conditions where recovery by the errant task is deemed to be impossible. The language defines some operations in which abortion must be deferred [BW98, Section 10.2.1]. In addition, the execution of some critical points of the run-time must be also deferred to keep its state stable. For this purpose the GNAT run-time uses a pair of subprograms (Abort_Defer, Abort_Undefer) that are called by the code expanded by the front-end to bracket unabortable operations involving task termination, (cf. Section 9.5), rendezvous statements (cf. Chapter 10), and protected objects (cf.

Chapter 11). The implementation of these primitives will be discussed in detail in Chapter 20.

14.4 Run-Time Subprograms for Task Creation and Termination

Section 9.6 discussed the sequence of calls to the run-time issued by the expanded code at the point of tasks creation and finalization. Figure 14.3 represents this sequence. Each rectangle represents a subprogram; the rhombus represents the new task.

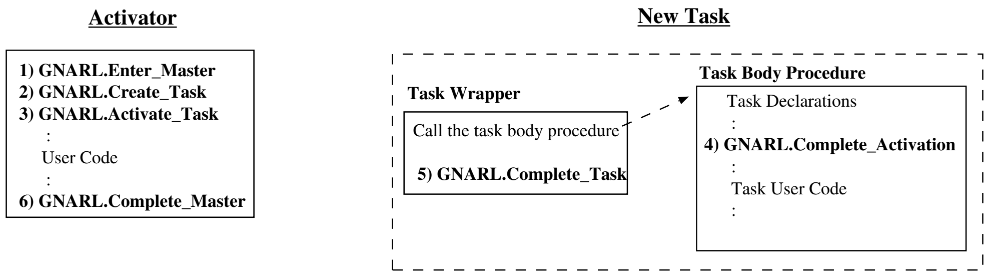


Figure 14.3: GNARL Subprograms Called During the Task Life-Cycle

The whole sequence is as follows:

1. *Enter_Master* is called in the Ada scope where a task or an allocator designating objects containing tasks is declared.
2. *Create_Task* is called to create the ATCBs of the new tasks and to insert them in the all tasks list and in the activation chain (see section 14.4.3).
3. *Activate_Tasks* is called to create new threads and to associate them with the new ATCBs on the activation chain. When all the threads have been created the activator becomes blocked until they complete the elaboration of their declarative part.

The thread associated with the new task executes a *task-wrapper* procedure. This procedure has some locally declared objects that serve as per-task run-time local data. The task-wrapper calls the *Task Body Procedure* (the procedure generated by the compiler which has the task user code) which elaborates the declarations within the task declarative part, to set up the local environment in which it will later execute its sequence of statements. Note that if these declarations also have task objects, then there is a chained activation: this task becomes the activator of dependent task objects and cannot start the execution of its user code until all dependent tasks complete their own activation.

4. *Complete_Activation* is called when the new thread completes the elaboration of all the task declarations, but before executing the first statetement in the task body. This call is used to signal to the activator that it need no longer wait for this task to finish activation. If this is the last task on the activation list to complete its activation, the activator becomes unblocked.

From here on the activator and the new tasks proceed concurrently and their execution is controlled by the POSIX scheduler. Afterward, any of them can terminate its execution and therefore the following two steps can be interchanged.

5. *Complete_Task* is called when a task terminates its execution. This may happen as a result of reaching the end of its sequence of statements, or by other means, such as an exception or an abort statement (cf. Chapter 20). Even though a completed task cannot execute any more, it is not yet safe to deallocate its working storage at this point because some reference may still be made to the task. In particular, it is possible for other tasks to still attempt entry calls to a terminated task, to abort it, and to interrogate its status via the *'Terminated* and *'Callable* attributes. Nevertheless, completion of a task does requires action by the run-time. The task must be removed from any queues on which it may happen to be, and must be marked as completed. A check must be made for pending calls on entries of the completed task, and the exception *Tasking_Error* must be raised in any such calling tasks [BR85, Section 4].

6. *Complete_Master* is called by the activator when it finishes the execution of its statements. At this point the activator waits until all its dependent tasks either complete their execution (and call *Complete_Task*) or are blocked in a *Terminate* alternative. Alive dependent tasks in a **terminate** alternative are then forced to terminate.

   In general this is the earliest point at which it is completely safe to discard all storage associated with dependent tasks (because it is at this point that execution leaves the scope of the task's declaration, and it is no longer possible for any dependent task to be awakened again by a call).

In the following sections we give more a detailed description of the work done by the following run-time subprograms: Enter_Master, Create_Task, Activate_Tasks, Complete_Activation, Complete_Task, and Complete_Master, which implement the most important aspects of tasking.

14.4.1 GNARL.Enter_Master

Enter_Master increments the current value of *Master_Within* in the activator.

14.4.2 GNARL.Create_Task

Create_Task carries out the following actions:

1. If no priority was specified for the new task then assign to it the base priority of the activator. When no priority is specified, the priority of a task is the priority at which it is created, that is, the priority of the activator at the point it calls *Create_Task*.
2. Traverse the parents list of the activator to locate the parent of the new task via the master level (the Parent Master is lower than the master of the new task).
3. Defer abortion.
    4. Request dynamic memory for the new ATCB (*New_ATCB*).
    5. Lock *All_Tasks_List* because this lock is used by Abort_Dependents and Abort_Tasks and, up to this point, it is possible for the new task is to be part of a family of tasks that is being aborted.
        6. Lock the Activator's ATCB.
            7. If the Activator has been aborted then unlock the previous locks (*All_Tasks_Lists* and its ATCB), undefer the abortion and raise the *Abort_Signal* internal exception.
            8. Initialize all the fields of the new ATCB: *Callable* set to True; *Wait_Count*, *Alive_Count* and *Awake_Count* set to 0 (cf. *System.Tasking.Initialize_ATCB*).
        9. Unlock the Activator's ATCB.
    10. Unlock *All_Tasks_List*.

11. Add some data to the new ATCB to manage exceptions (cf. *System.Soft_Links.Create_TSD*).
12. Insert the new ATCB in the activation chain.
13. Initialize the structures associated with the task attributes.

14. Undefer the abortion.

From this point the new task becomes callable. When the call to this run-time subprogram returns, the code generated by the compiler sets to *True* the variable which indicates that the task has been elaborated.

14.4.3 GNARL.Activate_Tasks

With respect to task activation the Ada Reference Manual says that all tasks created by the elaboration of object_declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together [AAR95, Section 9.2(2)].

GNAT uses an auxiliary list (the *Activation List*) to achieve this semantics. In a first stage all the ATCBs are created and inserted in the two lists (*All Tasks* and *Activation* lists); in a second stage the Activation List is traversed and new threads of control are created and associated with the new ATCBs. Although ATCBs are inserted in both lists in LIFO order all activated tasks synchronize on the activators lock before they start their activation in priority order. The activation chain is not preserved once all its tasks have been activated.

Activate_Tasks performs the following actions:

1. Defer abortion.

2. Lock *All_Tasks_List* to prevent activated tasks from racing ahead before we finish activating all tasks in the *Activation Chain*.

3. Check that all task bodies have been elaborated. Raise *Program_Error* otherwise.

   For the activation of a task, the activator checks that the task_body is already elaborated. If two or more tasks are being activated together (see ARM 9.2), as the result of the elaboration of a declarative_part or the initialization of the object created by an allocator, this check is done for all of them before activating any.

Reason: As specified by AI-00149, the check is done by the activator, rather than by the task itself. If it were done by the task itself, it would be turned into a Tasking_Error in the activator, and the other tasks would still be activated [AAR95, Section 3.11(12)].

4. Reverse the activation chain so that tasks are activated in the order they were declared. This is not needed if priority-based scheduling is supported, since activated tasks synchronize on the activators lock before they start activating and so they should start activating in priority order.

5. For all tasks in the activation chain do the following actions:

   (a) Lock the task's parent.

   (b) Lock the task ATCB.

   (c) If the base priority of the new task is lower than the activator priority, raise its priority to the activator priority, because a task being activated inherits the active priority of its activator [AAR95, Section D.1(21)].

   (d) Create a new thread by means of `GNARL` call (cf. *Create_Task*) and associates it to the task wrapper. If the creation of the new thread fails, release the locks and set the caller ATCB field Activation_Failed to *True*.

   (e) Set the state of the new task to *Runnable*.

   (f) Initialize the counters of the new task (*Await_Count* and *Alive_Count* set to 1)

   (g) Increment the parent counters (*Await_Count* and *Alive_Count*).

   (h) If the parent is completing the master associated with this new task, increment the number of tasks that the master must wait for (*Wait_Count*).

   (i) Unlock the task ATCB.

   (j) Unlock the task's parent.

6. Lock the caller ATCB.

7. Set the activator state to *Activator Sleep*

8. Close the entries of the tasks that failed thread creation, and count those that have not finished activation.

9. Poll priority change and wait for the activated tasks to complete activation. While the caller is blocked POSIX releases the caller lock.

   Once all of these activations are complete, if the activation of any of the tasks has failed (typically due to the propagation of an exception), Tasking_Error is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any tasks aborted prior to completing their activation are ignored when determining whether to raise Tasking_Error [AAR95, Section 9.2(5)].

10. Set the activator state to *Runnable*.

11. Unlock the caller ATCB.

12. Remove the Activation Chain.

13. Undefer the abortion.

14. If some tasks activation failed then raise *Program_Error*. Tasking_Error is raised only once, even if two or more of the tasks being activated fail their activation [AAR95, Section 9.2(5b)].

14.4.4 GNARL.Tasks_Wrapper

The *task-wrapper* is a GNARL procedure that has some local objects that serve as per-task local data.

14.4.5 GNARL.Complete_Activation

Complete_Activation is called by each task when it completes the elaboration of its declarative part. It carries out the following actions:

1. Defer the abortion.

2. Lock the activator ATCB.

3. Lock self ATCB.

4. Remove dangling reference to the activator (since a task may outline its activator).

5. If the activator is in the *Activator_Sleep* State then decrement *Wait_Count* in the activator. If this is the last task to complete the activation in the Activation Chain, wake up the activator so it can check if all tasks have been activated.

6. Set the priority to the base priority value.

7. Undefer the abortion.

14.4.6 GNARL.Complete_Task

The Complete_Task subprogram performs the following single action:

1. Cancel queued entry calls.

From this point the task becomes not callable.

14.4.7 GNARL.Complete_Master

The run-time subprogram Complete_Master carries out the following actions:

1. Traverse all ATCBs counting how many active dependent tasks does this master currently have (and terminate all the still unactivated tasks). Store this value in *Wait_Count*.

2. Set the current state of the activator to *Master_Completion_Sleep*.

3. Wait until dependent tasks are all terminated or ready to terminate.

4. Set the current state of the activator to *Runnable*.

5. Force those tasks on terminate alternatives to terminate (by aborting them).

6. Count how many *active* dependent tasks does this master currently have. Store this value in *Wait_Count*.

7. Set the current state of the activator to *Master_Phase_2_Sleep_State*.

8. Wait for all counted tasks to terminate themselves.

9. Set the current state of the activator to *Runnable*.

10. Remove terminated tasks from the list of dependents and free their ATCB.

11. Decrement *Master_Within*

14.5 Summary

In this chapter we have examined the basic data structures used by the GNAT run-time to support Ada tasks, the task states used by GNARL, some of the compiler-generated code that invokes run-time actions, and the subprograms called by this generated code. To summarize again:

1. Each task has an associated Ada Task Control Block (ATCB).

2. THe (*All Tasks List*) holds the ATCBs of all tasks in the program

3. One auxiliary list is used to activate task objects created in the same Ada scope at the same time.

4. A construct that declares tasks is a Master for these tasks. A task can itself be a Master. The presence of Masters determines all actions relating to task finalization.

5. A task declaration is translated by the compiler into a limited record which is part of the ATCB; the Ada task body is translated into a procedure with intermixed calls to the RTS to manage the task body creation, activation, communication and finalization.

6. The environment task is responsible for initialization of the RTS and the execution the main subprogram. As a consequence, the environment task is also the activator of all library-level tasks.

Chapter 15

The Rendezvous

The *Rendezvous* is the basic mechanism for synchronization and communication of Ada tasks. The model of Ada is based on a client/server model of interaction. One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients). It does this by declaring one or more public *entries* in its task specification. A rendezvous is requested by one task making an entry call on an entry of another task. For the rendezvous to take place the called task must accept this entry call. During the rendezvous the calling task waits while the accepting task executes. When the accepting task ends the rendezvous both tasks are freed to continue their execution. In the case that more than one task is waiting on the same entry of a task, Ada requires the calls be accepted in first-in-first-out order. The run-time must maintain data structures to keep track of which tasks are waiting on entry calls, which entries they are calling, and in what order the calls on each entry of a task arrived.

A conditional entry call differs from an unconditional entry call in that the calling task need not wait unless the call can be accepted immediately. If the called task is ready to accept, execution proceeds as for an unconditional call. Otherwise, the calling task resumes execution without completion of a rendezvous. The syntax provides for execution to resume at different places, depending on whether any rendezvous took place. The efficient implementation of the conditional entry-call requires a simple test for whether the called task is ready to accept. This can be done in constant time if the run-time maintains an accept vector for each task, telling on which entries, if any, the task is ready to accept a call (See the expansion of this vector in Section 10.4.2). If the test fails, the run-time may return control immediately to the calling task. Otherwise, the actions are similar to those for the unconditional call.

The contents of this chapter are structured as follows: Section 15.1 presents the entry call record; Section 15.2 presents the implementation of the entry queues; Section 15.3 presents the stack required to give support to nested accept statements; Section 15.4 presents the run-time support for the selective accept statement. Finally, Section 15.5 describes the sequence of actions carried out by the GNARL subprograms that give support for entry-calls and for the accept-statements.

15.1 The Entry-Call Record

The GNAT run-time associates a record to each entry call: the *Entry Call Record*. It is used to group all the run-time information associated with the entry call. It includes the identifier of the called entry, the current state of the entry call, the links to the previous and next queued entry calls, etc. If the entry has parameters, the front-end groups all the parameters in a contiguous block (cf. Section 10.2.1, and the run-time saves the base address of this block in the *Uninterpreted_Data* field of the *Entry Call Record*. Figure 15.1 presents the GNAT run-time data structures used to handle an entry call to the entry E of the following task specification:

```
task T is
   entry E (Number : in Integer; Text : in String);
end T;
```

An entry-call can be in one of the following states:

- *Never Abortable*. The call is not abortable, and never can be. It is used for calls that are made in a abort deferred region [AAR95, Section 9.8(5-11,20)]).
- *Not Yet Abortable*. The call is not abortable, but may become abortable in the future.
- *Was Abortable*. The call is not abortable, but once was. The *Was_* versus *Not_Yet_* distinction is needed to decide whether it is OK to advance into the abortable part of an async. select stmt. That is allowed iff the mode is Now_ or Was_.
- *Now Abortable*. The call is abortable.

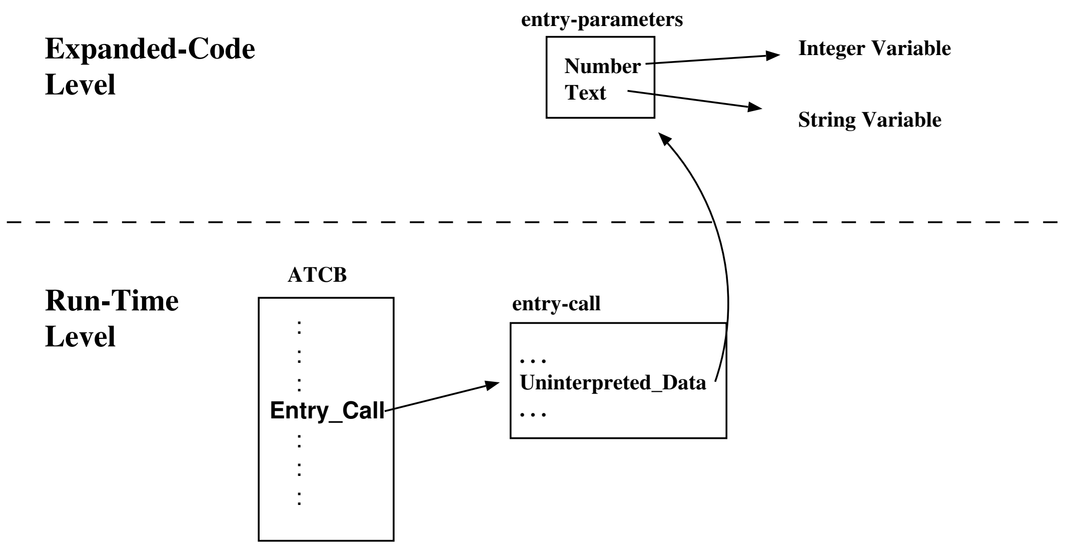


Figure 15.1: Data structures associated to an entry-call.

- *Done*. The call has been completed without cancellation, or no call has been made yet at this ATC nesting level (cf. Chapter 20, and so aborting the call is no longer an issue. Completion of the call does not necessarily indicate "success"; the call may be returning an exception if Exception_To_Raise is non-null.

- *Cancelled*. The call was asynchronous, and was cancelled.

15.2 Entries and Queues

Each entry has one queue which stores all the pending entry calls [AAR95, Section 9.1(16)]. If the queue is nonempty, the next caller to be served is at the head of the queue. The cost of checking whether there are any calls queued for a given entry depends on the data structure chosen for the entry queues. The GNARL run-time uses circular doubly linked lists so that checking, insertion and deletion are all constant-time operations.

The ATCB field *Entry_Queues* is an array indexed by the entry identifier (the front-end associates an unique identifier to each entry queue, cf. Section 10.1). Each element of this array has two fields: the *Head* and the *Tail* of the queue (cf. Figure 15.2).

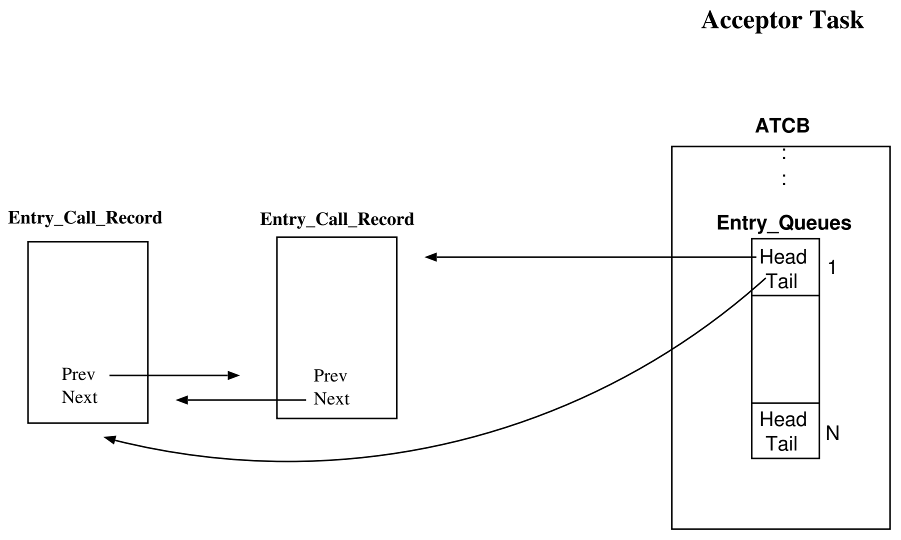


Figure 15.2: Entry Queues.

15.3 Accepted-Calls Stack

Because Ada allows the use of nested accept-statements, when an entry-call is accepted the GNAT run-time extracts the entry-call record from the corresponding entry-queue and pushes its address in an stack. The top of this stack is referenced by the *Call* field of the acceptor's ATCB (cf. Figure 15.3). The *Acceptor_Prev_Call* field links all the stack elements.

15.4 Selective Accept

The special implementation problem introduced by the selective wait is that a task may at one instant be ready to accept a call on a set of several entries. From the viewpoint of the Ada run-time, this is really two problems, since it comes up in the processing of entry calls, as well as selective waits:

1. Since a task may be waiting on more than one open accept alternative, processing an entry-call requires checking whether the called entry corresponds to one of the open alternatives.
2. Since there may be several open accept alternatives, processing the selective wait requires checking the set of pending entry calls against the set of open accept alternatives.

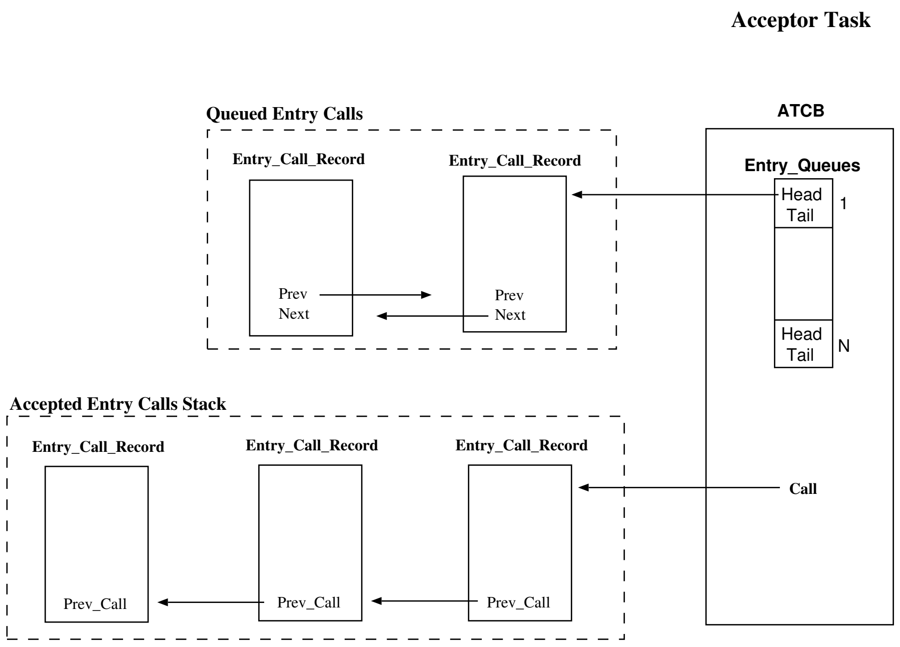


Figure 15.3: Simple Accept.

The need to be able to perform both of these operations efficiently strongly influences an implementation's choice of data structures. There are two obvious ways to perform the first operation, checking whether a called entry has a currently open accept alternative:

1.1. If the set of open accept alternatives is represented as a list, checking requires comparing the called entry against each of the entries in this list. We call this approach the use of an *open entry list*. It may be time consuming if there are many open entries.

1.2. An alternative is to use a vector representation for the set of open entries: the *open accepts vector*. This vector would have one component for each entry of the task. Each component would minimally indicate whether the corresponding entry is open.

Note that the accept vector or open entry list must be created at the time the selective wait statement is executed, once it is known which alternatives are open. The time needed to do this only depends on the number of alternatives in the selective wait statement. With separate queues for each entry, it is necessary to check the queue corresponding to each open entry. This requires sequencing through the open entries. Alternatively, if the open entries are represented by an open entry list, this check can be performed more quickly, without looking at the non-open entries. This may be a good reason to keep both an open entry list and an accept vector, though this redundancy may cost more in overhead than it saves through faster execution of the check for pending calls.

GNAT uses the *Open Accepts Vector*. Each element of this vector has two fields: the entry identifier and a boolean which indicates if the accept statement has a null body (cf. Section 10.4.2). Each element of the accept vector corresponds to the accept alternatives of the select statement (in the same order; first element of the accept vector corresponds to the first alternative, second element corresponds to the second alternative, etc.). The run-time returns 0 when the entry guard is closed.

15.5 Run-Time Rendezvous Subprograms

Chapter 10 presents the expansion of the entry-call and accept statements. The following sections describe the actions carried out by the GNAT run-time subprograms called by the expanded code.

15.5.1 GNARL.Call_Simple

The run-time subprogram *Call_Simple* simply delegates the work to other run-time subprogram called *Call_Synchronous*.

15.5.2 GNARL.Call_Synchronous

The run-time subprogram *Call_Synchronous* carries out the following actions:

1. Defer the abortion.
2. Create and elaborate a new entry-call record, and save on it the address of the parameters block.
3. Call the GNARL subprogram *Task_Do_Or_Queue*.
4. Wait for the completion of the rendezvous (*Wait_For_Completion*).
5. Undefer the abortion.

6. Raise any pending exception from the entry call (*Check_Exception*).

15.5.3 GNARL.Task_Do_Or_Queue

The subprogram *Task_Do_Or_Queue* carries out the following actions:

1. Try to serve the call immediately. If the acceptor is accepting some entry call and the current call can be accepted the following actions are carried out:
   (a) Commit the acceptor to rendezvous with the caller.
   (b) If the acceptor is in a **terminate** alternative then cancel the terminate alternative. If the acceptor has no dependent tasks notify its parent that the acceptor is again awake.
   (c) If the **accept** statement has a null body (an accept used for tasks synchronization) then wake up the acceptor, wake up the caller and RETURN.
   (d) If the **accept** statement has some body then call a run-time procedure (*Setup For Rendezvous With Body*) to insert the *Entry Call Record* in the accepted entry-calls stack of the acceptor task (cf. Section 15.3), and to raise the priority of the acceptor (if the caller priority is higher than the priority of the acceptor). Then wake up the acceptor and RETURN.

15.5.4 GNARL.Task_Entry_Call

If the entry-call can be immediately accepted *Task_Entry_Call* carries out the same actions of the simple-mode entry-call and sets one out-mode parameter to True (Successful) to indicate this to the expanded code (cf. Section 10.2.2). Otherwise it sets this parameter to False. The expanded code uses this parameter to select the part of the user-code which must be executed after the call. Note that in the call is never enqueued; a conditional entry-call is only enqueued if the acceptor requeues it not-abortably (by means of a requeue-statement).

15.5.5 GNARL.Accept_Trivial

GNARL.Accept_Trivial performs the following actions:

1. Defer the abortion.
2. If no entry call is still queued then block the acceptor task to wait for the next entry call (*Wait_For_Call*).
3. Extract the entry-call record from the head of the queue (*Dequeue Head*) and wake-up the entry caller (*Wakeup_Entry_Caller*).
4. Undefer the abortion.

15.5.6 GNARL.Accept_Call

The GNARL procedure *Accept_Call* carries out the following actions.

1. Defer the abortion.
2. If the entry has no queued entry calls then block the acceptor tasks to wait for the next entry call (*Wait_For_Call*).
3. Extract the entry-call record from the head of the queue (*Dequeue Head*) and push it in the accepted entry-calls stack.
4. Update the out-mode parameter *Param_Access* with the reference to the *Entry Parameters Record* so that the compiler generated code can access the entry parameters.
5. Undefer the abortion.

15.5.7 GNARL.Complete_Rendezvous

If no exception is raised during the execution of an accept body the subprogram *Complete_Rendezvous* is called is called by the expanded code. This subprogram just calls the subprogram *Exceptional_Complete_Rendezvous* notifying it that no exception was raised.

15.5.8 GNARL.Exceptional_Complete_Rendezvous

If an exception was raised during the execution of the code associated with the entry call, the exception must be also propagated on the caller and on the acceptor task [AAR95, Section 9.5.2]. For this purpose the subprogram *Exceptional_Complete_Rendezvous* carries out the following actions:

1. Defer the abortion.
2. Pop the reference to the entry-call record from the accepted entry-calls stack.
3. If an exception was raised, get its identifier from the entry call field *Exception_To_Raise* and save its occurrence in the ATCB field *Compiler_Data*. This exception will be propagated back to the caller when the rendezvous is completed [AAR95, Section 9.5.3].
4. Wake up the caller (*Wakeup_Entry_Caller*).
5. Undefer the abortion.

15.5.9 GNARL.Selective_Wait

The GNARL subprogram *Selective_Wait* performs the following actions:

1. Defer the abortion.
2. Try to serve the entry call immediately. GNARL subprogram *Select_Task_Entry_Call* selects one entry call following the queuing policy being used.
   (a) If there is some candidate and the accept has a null body then complete the rendezvous, wake up the caller, undefer the abortion and RETURN.
   (b) If there is some candidate and the accept has some associated code then insert the entry-call record in the accepted entry-calls stack (*Setup_For_Rendezvous_With_Body*), update the reference to the parameters-block, undefer the abortion and RETURN.
   (c) If there is no candidate but there are alternatives opened, wait for a caller. In the future some caller will put an entry call record in the accepted entry-calls stack and it will wake up this acceptor. Then this acceptor will update the reference to the entry parameters, it will undefer the abortion, and it will RETURN.

(d) If there is a terminate alternative, notify its ancestors that this task is on a terminate alternative (*Make_Passive*, and wait for normal entry call or termination.

(e) If no alternative is open and no delay (or terminate) has been specified then raise the predefined exception *Program_Error*.

15.5.10 GNARL.Task_Count

The function *Task_Count* gives support to the 'Count attribute. It returns the number of queued entry calls in the specified entry queue.

15.6 Summary

The *Rendezvous* is the basic mechanism for synchronization and communication of Ada tasks. In this chapter, the main aspects of the GNAT implementation have been described. In summary:

- The run-time information associated with the entry call is grouped into an *Entry Call Record*.
- The compiler generates one *Entry Parameters Record* with the address of the real-parameters. GNARL registers the address of this record in a field of the Entry Call Record.
- The entry queues are implemented by means of double linked lists of Entry Call Records.
- Nested accepts are handled by means of one accepted entry-calls Stack; a linked list of accepted Entry Call Records.
- An *Accept Vector* is used to evaluate the open guards of the selective accept.

Chapter 16

Protected Objects

The high burden of threads synchronization required by the Ada rendezvous was inappropriate for the implementation of Systems with fast response-time requirements. For this reason, Ada 95 has a more efficient tasking synchronization mechanism based on shared memory: the Protected Objects. Protected procedures and entries must be executed under read/write locks; however, because protected functions are not permitted to affect the state of the protected object, they can be execute under read-only locks, which permits an implementation to execute several calls to protected functions in parallel.

To issue a call to a protected object, a task simply names the object and the required subprogram or entry. As with task entry calls, the caller can use the select statement to issue a *timed* or *conditional* entry call. Clearly, it is possible for more than one task to be queued on a particular protected entry. As with task queues, a protected entry is, by default, ordered in a first-in-first-out fashion; however, if the Real-Time Systems Annex is being supported, other queuing disciplines are allowed. When a call on a protected procedure or protected entry is executed, the barrier is evaluated; if the barrier is closed (evaluates to False), the call is queued. Any exception raised during the evaluation of a barrier results in *Program_Error* being raised in all tasks currently waiting on the entry queues associated with the protected object containing the barrier [BW98, Chapter 7.8]).

When the execution of a protected procedure or entry is completed, all the barriers are re-evaluated and, potentially, entry bodies are executed. After executing the body of one protected procedure or entry all the PO barriers which have queued tasks are reevaluated. If some entry is now open the entry call is accepted and the corresponding entry body is executed. This process repeats until there is no barrier with queued tasks open. If several barriers are open after the execution of a protected procedure or entry Ada does not specify which entry is then serviced.

Chapter 11 not only discusses the expansion of protected-type, barriers, and protected subprograms, but also presents the two main implementation models for protected objects, self-service and proxy, as well as the proposed implementations (cf. Section 11.1). As it is discussed there, GNAT follows the call-back implementation of the proxy model. One of the main reasons (from the viewpoint of the run-time) is that using Pthreads to implement the self-service model introduces one important problem: The task attempting to exit an eggshell must be able to transfer ownership to a task waiting on an *Open* entry. However, there is no good way to solve it with Pthreads because, though it is possible to force a thread to be given a mutex by raising its priority over that of the other contenders, this may lead to unnecessary context switches and degrades the implementation of Ada priority over Pthreads.

16.1 The Lock

According to Ada semantics, a queued entry call has precedence over other operations on the protected object. This is often explained in terms of the *eggshell model*. The lock on a protected object is the eggshell. Figure 16.1 is a graphical representation of the protected objects. Threads are represented by shadowed circles; the two levels of the protected objects are represented by means of a big circle (associated with the object lock) and a big rectangle (associated with the object state and operations); small rectangles represent the protected operations: black rectangles represent closed entries and white rectangles represent open entries. Accordingly, this example presents one thread executing a protected operation (it is inside the PO), two threads queued in a closed entry, one thread queued in the entry which is now being executed under mutual-exclusion, and some additional threads which are not queued.

16.2 Run-Time Subprograms

16.2.1 GNARL.Protected_Entry_Call

A simple call to a protected entry is expanded by the front-end into a call to the GNARL subprogram *Protected_Entry_Call*. The entry-call is handled by the run-

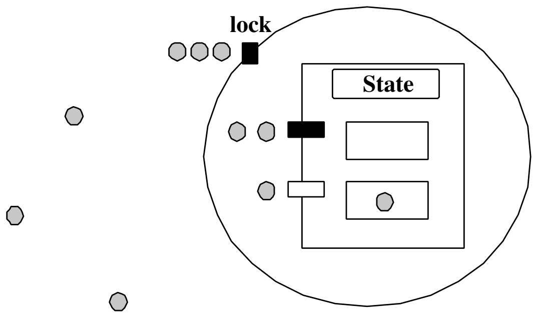


Figure 16.1: Graphical Representation of the Protected Object.

time similar to task entry-calls (cf. Section 15.5.4). This facilitates the implementation of the Ada requeue statement. Its whole sequence of actions is as follows:

1. Defer the abortion.
2. Write lock the object.
3. Elaborate a new *Entry Call Record*.
4. Call the GNARL procedure *PO_Or_Queue* to issue the call or to enqueue it in the corresponding entry queue.
5. Call the GNARL procedure *PO_Service_Entries* to service the opened entries.
6. Unlock the object.
7. Undefer the abortion.
8. Check if some exception must be re-raised.

In case of conditional and timed entry calls, the actions carried out by the GNAT run-time are basically the sequence presented above. However, if the barrier is closed the entry-call record is not enqueued, and the run-time sets to 0 the index of the selected entry. This value is used by the expanded code to execute the *else part* of the conditional entry call.

16.2.2 GNARL.PO_Do_Or_Queue

The sequence of actions carried out by *PO_Do_Or_Queue* is as follows:

1. Call the barrier function.
2. If the barrier is closed then enqueue the *Entry Call Record*, and RETURN.
3. If the barrier is open then execute the steps 2 to 9 of the GNARL procedure *Service_Entries*.

16.2.3 GNARL.Service_Entries

The run-time must evaluate the entry-barriers after executing a protected procedure or entry, essentially treating the barrier expression as though they depended only on the state of the protected object. In the self-service model, only when the barriers of all entries with queued calls are *False* the thread can leave the eggshell. This assures that all entry calls made eligible by a state change are executed before any further operations are initiated. For this purpose, the front-end expands the entry barriers and bodies into functions and procedures, and generates a table initialized with their addresses. The run-time receives this table and uses the pointers to call the functions which evaluate the entry barriers, and to call the corresponding body when the barrier is open.

The basic algorithm of the GNARL *Service_Entries* procedure is as follows:

```
 1 while <There_Is_Some_Open_Barrier_With_Queued_Entry_Calls> loop
 2    Update object reference to the Entry_Call_Record
 3    begin
 4      Call the Entry_Body
 5    exception
 6      when others => Broadcast Program_Error
 7    end
 8    Remove the Reference to the Entry_Call_Record
 9    GNARL.Wake_Up_Entry_Caller
10 end loop;
```

Line 1 is evaluated by the GNARL procedure *Select_Protected_Entry_Call* which traverses all the entry queues and reevaluates the barrier of those entries with queued entry calls. As soon as some barrier is open (it evaluates to true), GNARL selects it to be serviced. In line 2, the *Call_In_Progress* field of the *_object* (see the *Protection_Entries* type definition) is set to the selected entry call record to remember that this is the entry call being attended. Lines 3 to 7 open a new scope to issue the call to the entry body and to handle the exceptions in the user code. In this case the predefined exception *Program_Error* is broadcasted to all tasks currently queued in any entry of the protected object. In line 8 the reference to the entry call is removed (this entry call has been attended) and the task entry caller is woken up (line 9). After this work the loop is again executed and the entry barriers are reevaluated. This process stops when no open barrier is found in an entry with queued tasks.

16.3 Summary

In this chapter we have briefly presented the sequence of actions carried out by the run-time subprograms which give support to protected subprograms.

Chapter 17

Time and Clocks

Tasks can delay their execution for a period of time, or until an absolute time is reached. In both cases this enables the task to be queued on some future event rather than busy-wait o calls to the clock function. Tasks can also issue timed entry-calls. If the call is not accepted before the expiration of the specified delay, the run-time must cancel the entry-call wake-up the calling task. In addition, the timed selective accept allows a server task to time-out if an entry call is not received within a certain period of time.

Ada gives access to the clock by providing two packages: *Calendar* and *Real_Time*. Calendar provides an abstraction for "wall clock" time that recognizes leap years, leap seconds and other adjustments; Real_Time gives a second representation that defines a monotonic (that is, non-decreasing) regular clock. Although these representations map down to the same hardware clock, they cater for different application needs.

17.1 *Delay* and *Delay Until* Statements

The GNARL subprograms which implement these Ada statements are placed in child packages of the corresponding standard Ada packages: *Ada.Calendar.Delays* and *Ada.Real_Time.Delays*. The GNAT front-end expands a delay-statement into a call to the corresponding GNARL subprogram.

GNARL provides two implementations of the delay statements: one for the case of an Ada program without tasks and the other for an Ada program with tasks. A link is used to access the proper subprogram (*Timed_Delay*).

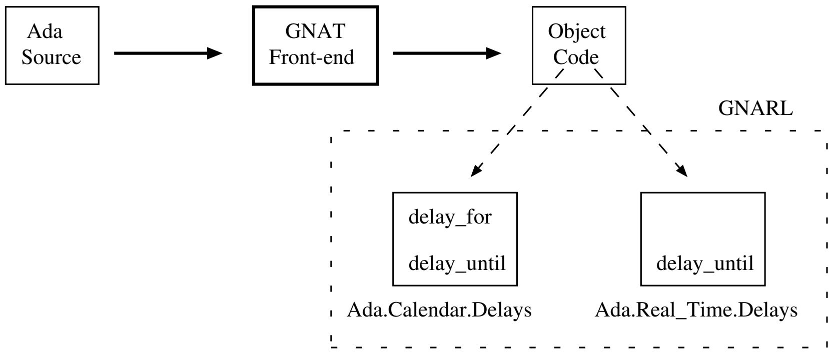


Figure 17.1: GNARL Subprograms for the Delay Statement.

- In case of no tasking this link points to the GNARL procedure *Time_Delay_NT*, which calls the GNULL procedure *Timed_Delay* (cf. Figure 17.2).

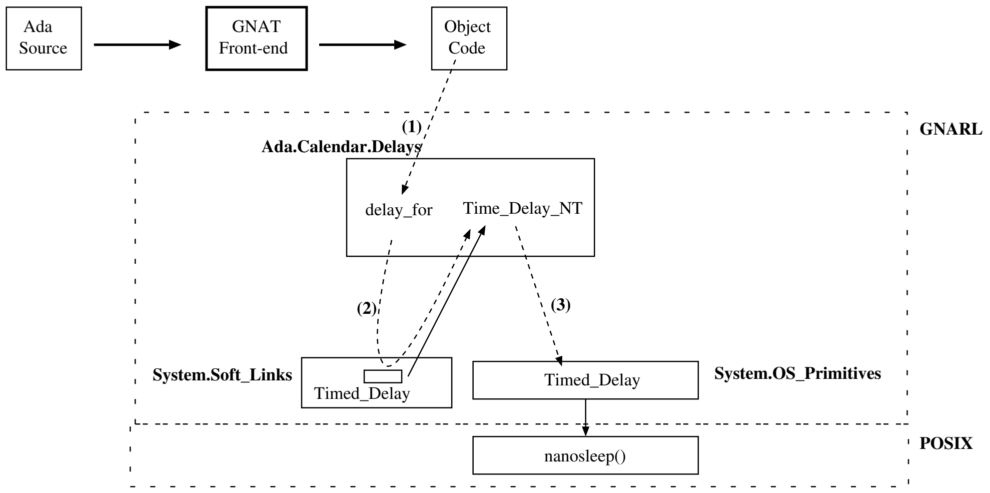


Figure 17.2: GNARL Subprograms for the Delay Statement in an Ada Program without Tasks.

- In case of a program with tasks this link points to the GNARL procedure *Timed_Delay_T*, which calls another version of the GNULL procedure *Timed_Delay* (cf Figure 17.3).

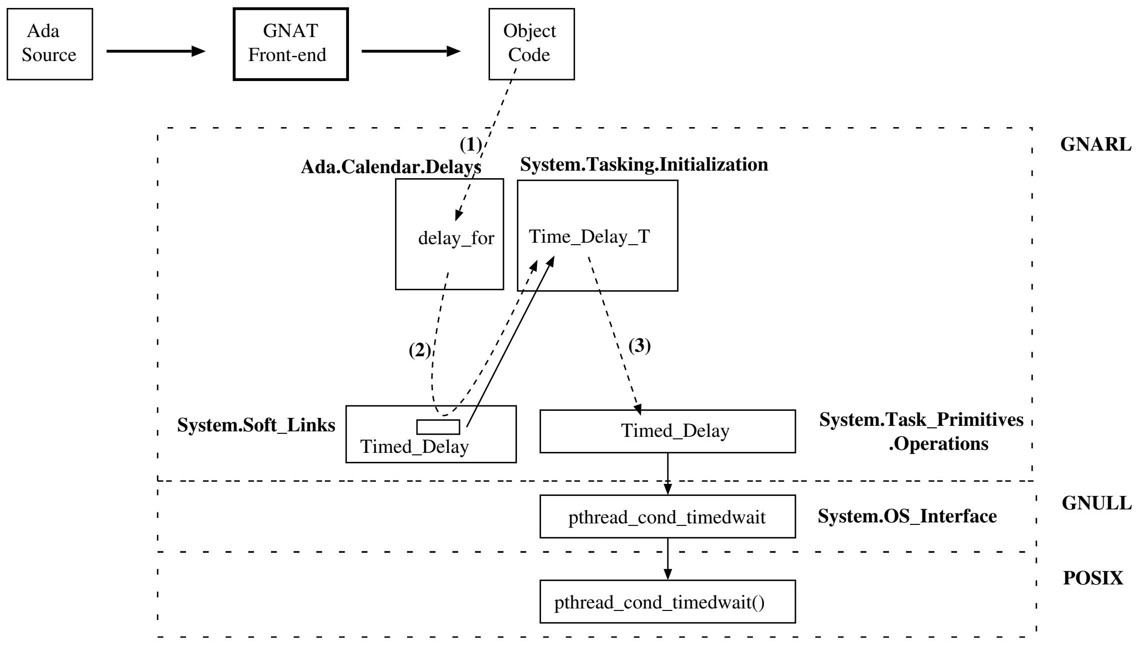


Figure 17.3: GNARL Subprograms for the Delay Statement in an Ada Program with Tasks.

17.2 Timed Entry Call

The timed task entry call is handled by the GNAT compiler in a similar way to the simple mode entry call (described in section 10.2.1). The compiler generates a call to the GNARL subprogram *Timed_Task_Entry_Call*. Basically this procedure carries out the same actions described in the simple mode entry call (section 15.5.1). However, if the entry can not be immediately accepted, it does not simply block the caller; it calls another GNARL subprogram to arm a timer and block the caller until the timeout expires. Figure 17.4 shows the GNARL and GNULL subprograms involved in this action. If the entry call is accepted before this timer expires, the timer is un-armed; otherwise the entry call is removed from the queue.

The GNAT implementation of the timed protected entry call follows the same scheme described above. However, the only difference is that the compiler generates a call to the GNARL procedure *Timed_Protected_Entry_Call*.

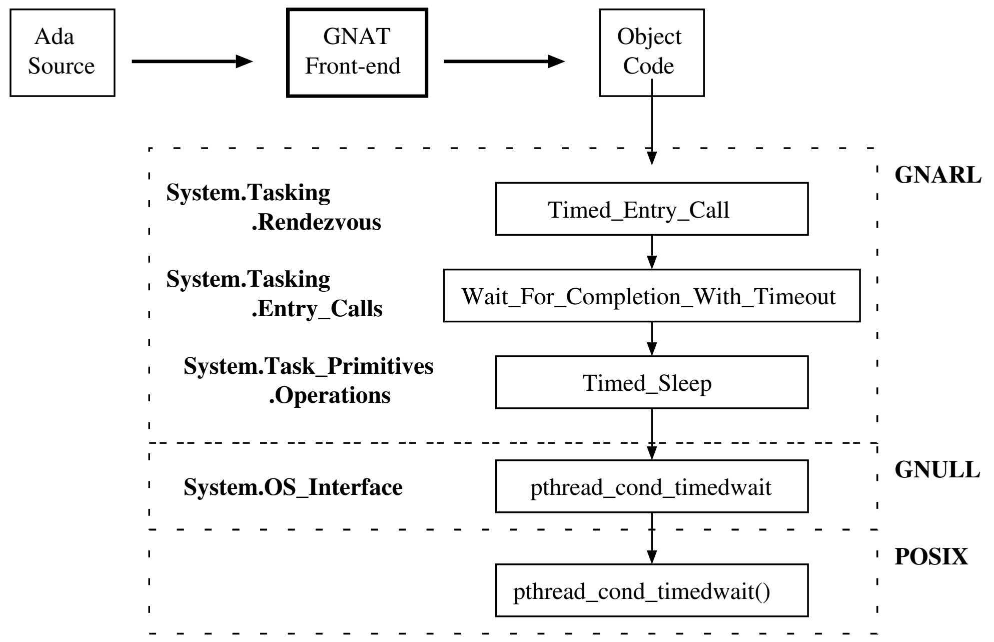


Figure 17.4: GNARL Subprograms for Timed Entry Call.

17.3 Timed Selective Accept

The timed task entry call is handled by the GNAT compiler in a similar way to the selective accept (described in section 15.4). The compiler generates a call to the GNARL subprogram *Timed_Selective_Wait*. Basically this procedure carries out the same actions described in case of the selective wait (section 15.5.9). However, if there is no entry call that can be immediately accepted, it does not simply block the caller; it calls another GNARL subprogram to program a timer and block the caller until this timeout expires. Figure 17.5 shows the GNARL and GNULL subprograms involved in this action. If some entry call is received before this timer expires, the timer is un-armed; otherwise the statements after the **delay** sentence are executed.

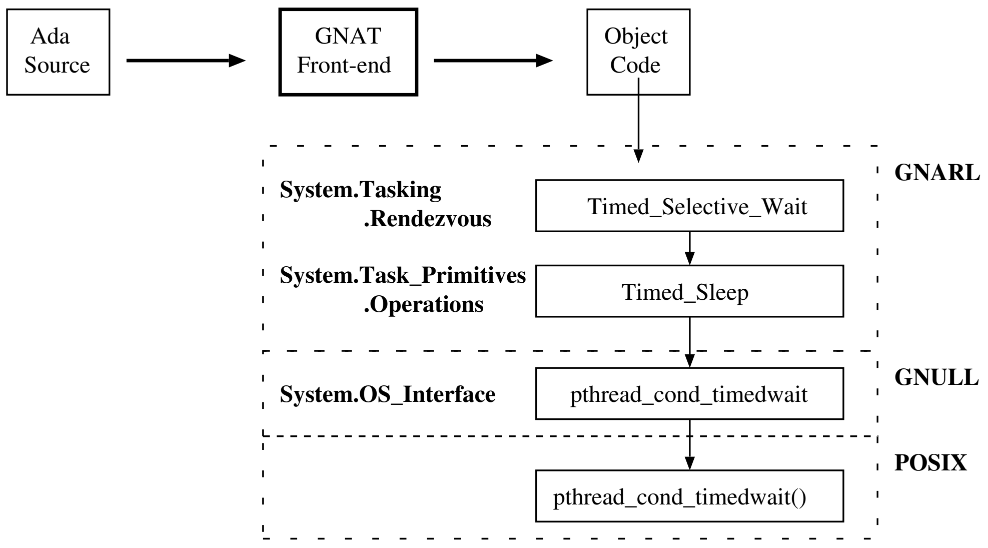


Figure 17.5: GNARL Subprograms for Timed Selective Accept.

17.4 Run-Time Subprograms

17.4.1 GNARL.Timed_Delay

When the program has tasks, the GNARL procedure *Timed_Delay* performs the following actions.

1. Defer the abortion.
2. Lock the ATCB of the calling task.
3. If the specified delay is a relative time span (that is, a **delay** statement), this delay it is converted to absolute time span by adding the current value of the clock.
4. If the specified time is a future time then
   (a) Set the state of the calling task to *Delay_Sleep*.
   (b) Call the POSIX function *pthread_cond_timedwait* to suspend the calling tasks until the specified time.
   (c) Set the state of the calling task to *Runnable*.
5. Unlock the ATCB of the calling task.

6. Yield the processor (this ensures that *"a delay statement always corresponds to at least one task dispatching point"* [AAR95, Section D.2.2(18)].

7. Undefer the abortion.

17.5 Summary

GNAT provides two implementations for the simple **delay** and **delay until** Ada sentences: one for the Ada programs without tasks, and another for the Ada programs with tasks. An access to a procedure is used to avoid multiple checks in the run-time to call the appropriate subprogram.

A timed entry call allows the task that executes it to make an entry call with the provision that it be awakened and the call canceled, if the call is not accepted before the expiration of a specified delay. As with the conditional entry call, provision is made for execution to resume in different places, depending on whether a rendezvous takes place. In addition to the processing required for a normal entry call, the timed entry call requires scheduling of a wake-up event if the call cannot be accepted immediately. If the call is accepted before this delay expires, the calling task must be removed from the delay queue. If the delay expires first, the task must be removed from the entry queue.

The GNAT implementation of the timed entry call sentences (to a protected entry or to a task entry) and the timed selective accept follow the same steps of the non-timed cases, though a timer is activated when the caller becomes blocked.

Chapter 18

Exceptions

An exception represents a kind of exceptional situation; an occurrence of such a situation (at run-time) is called an *Exception Occurrence*. When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned* and the corresponding exception-handler is executed. If the construct had no exception-handler, then the exception ocurrence is propagated. To *propagate* an exception occurrence is to raise it again in the innermost dynamically enclosing execution context [AAR95, Section 11-4]. If an exception occurrence is unhanded in a task body, the exception does not propagate further (because there is no dynamically enclosing execution). If the exception occurred during the activation of the task, then the activator would raise *Tasking_Error* [AAR95, Section 11-4]. There is a predefined library which provides additional facilities for exceptions (Ada.Exceptions).

18.1 Data Structures

18.1.1 Exception_Id

Each distinct exception is represented by a distinct value of type *Exception_Id*. The special value *Null_Id* does not represent any exception, and is the default initial value of the type *Exception_Id*. Each occurrence of an exception is represented by a value of the type *Exception_Occurrence*. Similarly, *Null_Occurrence* does not represent any exception occurrence; it is the default initial value of type *Exception_Occurrence*.

The GNAT run-time implements the exception identifier as an access to a record (*Exception_Data_Ptr*). Figure 18.1 presents the fields of this record. The field *Not_Handled_By_Others* is used to differentiate the user-defined exception from the run-time internal exceptions (i.e. task abortion) which can not be handled by the user-defined exception handlers. The field *Lang* defines the language where the exception is declared (by default "Ada"). The next two fields are used to store the full name of the exception. This name is composed of a prefix (the full path of the scope where the exception is declared) and the exception name. The last field is used to create linked lists of exception identifiers (described in section 18.1.2).

Exception_Data

Not_Handled_By_Others	: Boolean
Lang	: String (1 .. 3)
Name_Length	: Natural
Full_Name	: String_Ptr
HTable_Ptr	: Exception_Data_Ptr

Figure 18.1: Exception Identifier Record

When an exception is raised, the corresponding exception occurrence is stored by the GNAT run-time in the *Compiler_Data* field of the ATCB. The data type of this field is a record; the *Current_Excep* field of this record saves the exception occurrence.

The *Exception_Raised* field is set to *True* to indicate that this exception occurrence has actually been raised. When an exception occurrence is first created, it is set to false; then, when it is later processed by the GNARL subprogram *Raise_Current_Exception*, it is set to *True*. This allows the run-time to distinguish if it is dealing with an exception re-raise.

18.1.2 The Exceptions Table

Because the visibility rules of Ada exceptions (an exception may not be visible, though handled by the **others** handler, re-raised and then again visible to some other calling scope) a global table must be used (*Exceptions_Table*). In order to handle the exceptions in an efficient way, the Ada run-time uses a hash table (cf. Figure 18.3).

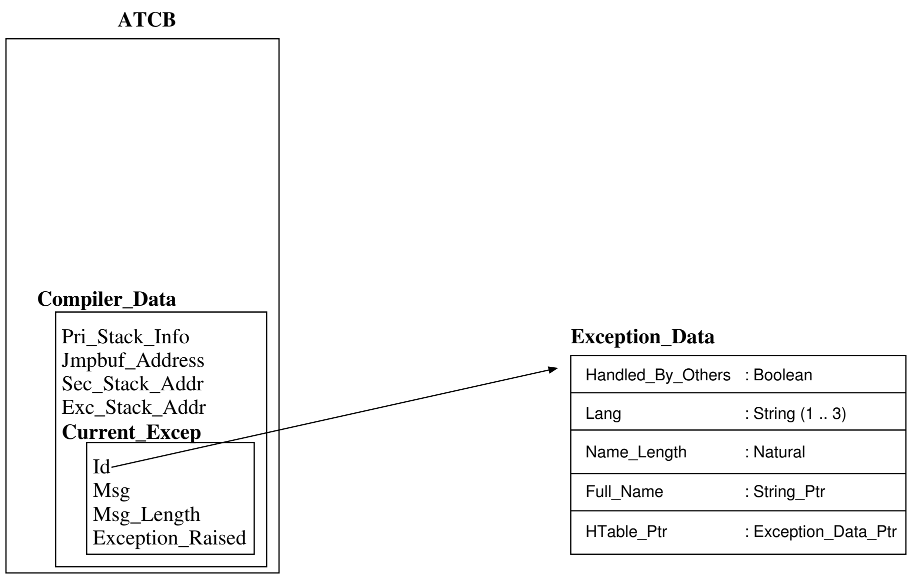


Figure 18.2: Occurrence Identifier.

As the reader can see, an accesses table to the exception identifiers is used. A simple linked list of exception identifiers is used to handle collisions. The field *HTable_Ptr* is used to link the exception identifiers.

When an exception is raised in a task, the corresponding exception identifier must be found. Therefore the hash function is evaluated, and the resulting linked list is traversed to look for the exception identifier. Then its reference is stored in the ATCB of the task. This reference is kept in the ATCB until the exception is handled (though the exception may not be visible in some exception handlers).

18.2 Run-Time Subprograms

18.2.1 GNARL.Raise

Ada allows an exception to be raised in two different ways: (1) by means of the **raise** statement and (2) by means of the procedure *Ada.Exceptions.Raise_Exception* which allows the programmer to associate a message to the exception. In both cases, the compiler generates a call to a GNARL function which carries out the following actions:

1. To fill the ATCB exception occurrence.

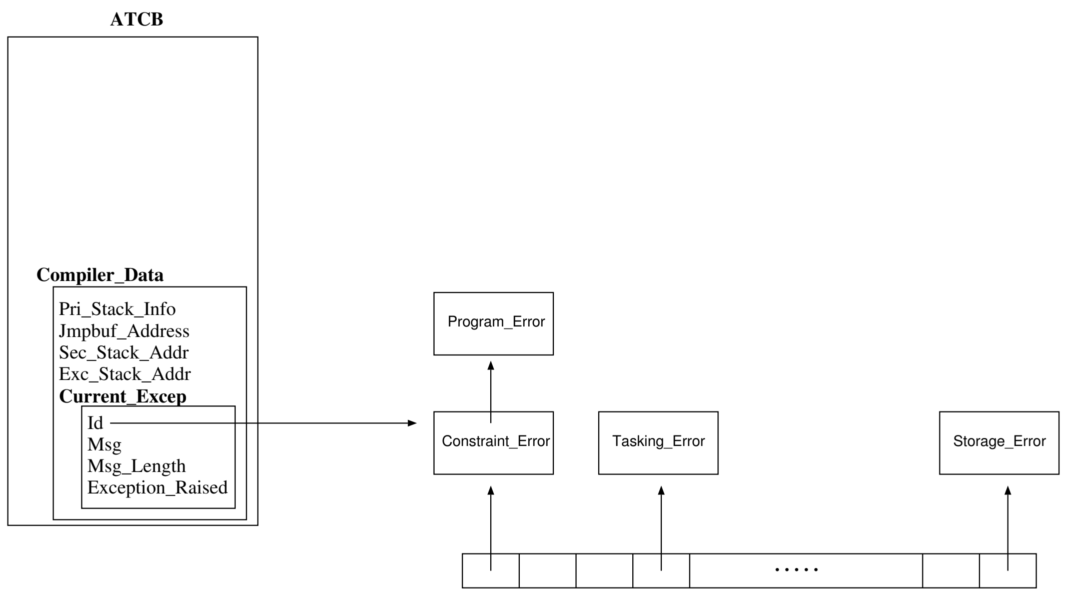


Figure 18.3: Hash Table.

2. To defer the abortion.
3. If there is one exception handler installed, then jump to it.
4. Otherwise (no exception handler can be called) terminate the execution of the program.

18.3 Summary

In this chapter the basic concepts of the GNAT exception handling implementation has been presented. The exception ID is an access to a record where the full name of the exception is stored. The exception occurrence is stored in the ATCB. All the exceptions are stored in a hash table.

Chapter 19

Interrupts

An interrupt represents a class of events that are detected by the hardware or system software. The occurrence of an interrupt consists of its generation and its delivery: the generation of an interrupt is the event in the underlying hardware or system which makes the interrupt available to the program; delivery is the action which invokes a part of the program (called the interrupt handler) in response to the interrupt occurrence. In between the generation of the interrupt and its delivery, the interrupt is said to be pending. The handler is invoked once for each delivery of the interrupt. While an interrupt is being handled, further interrupts from the same source are blocked; all future occurrences of the interrupt are prevented from being generated. It is usually device dependent as to whether a blocked interrupt remains pending or is lost.

Ada allows to associate an interrupt to a protected procedure or a task entry declared at library level. However, the association of a task entry is considered an obsolescent feature of the language [AAR95, Section J.7]. For this reason, in this chapter we will focus our attention on user-defined protected-procedure interrupt-handlers.

Certain interrupts are reserved. The programmer is not allowed to provide a handler for a reserved interrupt. Usually, a reserved interrupt is handled directly by the Ada run-time (for example, a clock interrupt used to implement the delay statement). Each non-reserved interrupt has a default handler that is assigned by the run-time system.

There are two two styles of interrupt-handler installation and removal: nested and non-nested. In the nested style, an interrupt handler in a given protected object is implicitly installed when the protected object comes into existence, and the treatment that had been in effect beforehand is implicitly restored when the protected object ceases to exist. In the non-nested style, interrupt handlers are installed explicitly by procedure calls, and handlers that are replaced are not restored except by explicit request [Coh96, Section 19.6.1].

The front-end identifies a handler to be installed in the nested style because it must have the pragma *Attach_Handler* specifying the corresponding interrupt_id. Dynamic allocation of protected objects gives greater flexibility. Allocating a protected object with an interrupt handler installs the handler associated with that object, and deallocating the protected object restores the handler previously in effect. Similarly, a handler to be installed in the non-nested style is identified by pragma *Interrupt_Handler*. This pragma imposes a restriction on the object: it must be dynamically created [Coh96, Section 19.6.1]. Non-nested installation and removal of interrupt handlers relies on additional facilities of package *Ada.Interrupts* [AAR95, Section C.3(2)].

To foster a simple, efficient and multi-platform implementation, GNAT reuses the POSIX support for signals and adds the minimum set of run-time subprograms required to achieve the Ada semantics. This work is simplified because POSIX signals are delivered to individual threads in a multi-threaded process using much of the same semantics as for delivery to a single-threaded process [GB92, Section 5.1].

19.1 POSIX Signals

A POSIX signal is a form of software interrupt which can be generated in several ways. A signal may be generated [DIBM96, Section 2]:

- By a hardware trap including division by zero, a floating-point overflow, a memory protection violation, a reference to a non-existent memory location or an attempt to execute an illegal instruction.
- Because a clock reaches a specified time, or a specified span of time has elapsed.
- By an asynchronous operation. Asynchronous input and output operations generate a signal when an operation completes, or if an operation fails.
- Because the user hits certain keys on the terminal that is controlling the process. Certain keys sequences allow the user to suspend, resume and terminate the execution of a process via signals.

- By a POSIX thread. POSIX threads may send a signal to another POSIX thread in the same process to notify it of an event, by calling *pthread_kill*.

Each POSIX thread has a signal mask: when a signal is generated for a thread and the thread has the signal masked, the signal remains pending until the thread unmasks it; the interface for manipulating the thread signal mask is *pthread_sigmask*. Only one pending instance of a masked signal is required to be retained; that is, if a signal is generated *N* times while it is masked the number of signal instances that are delivered to the thread when it finally unmasks the signal may be any number between 1 and N.

Each POSIX signal is associated with some *action*. The action may be to ignore the signal, terminate the process, continue the process, or execute a call to user-defined handler function (asynchronously and preemptively with respect to normal execution of the process). POSIX.1 specifies a default action for each signal. For most signals the application may override the default action by calling the function *sigaction*. The use of asynchronous handler procedures for signals is not recommended for POSIX threads, because the POSIX thread synchronization operations are not safe to be called within an asynchronous signal handler; instead, POSIX.1c recommends use of the *pthread_sigwait* function, which "accepts" one of a specified set of masked signals.

19.1.1 Reserved Signals

The definitions of "reserved" differs slightly between the ARM and POSIX. ARM specifies [AAR95, Section C.3(1)]:

The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. Program unit can be connected to non-reserved interrupts.

POSIX.5b/.5c specifies further [s-intman.adb]:

Signals which the application cannot accept, and for which the application cannot modify the signal action or masking, because the signals are reserved for use by the Ada language implementation. The reserved signals defined by this standard are:

- *Signal_Abort*
- *Signal_Alarm*
- *Signal_Floating_Point_Error*
- *Signal_Illegal_Instruction*
- *Signal_Segmentation_Violation*
- *Signal_Bus_Error*

If the implementation supports any signals besides those defined by this standard, the implementation may also reserve some of those.

The signals defined by POSIX.5b/5c that are not specified as being reserved are SIGHUP, SIGINT, SIGPIPE, SIGQUIT, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, SIGIO, SIGURG and all the real-time signals.

The GNAT FSU Linux implementation handles 32 signals. In this case the reserved signals are:

Number	Name	REASON	Description
2	* SIGINT	GNAT	Abort (used for CTRL-C)
4	* SIGILL	POSIX (HW)	Illegal Instruction
5	* SIGTRAP	GNAT	Trace trap
6	* SIGABRT	GNAT	Tasks abortion
7	* SIGBUS	POSIX (HW)	Bus error
8	* SIGFPE	POSIX (HW)	Floating Point Exception
9	SIGKILL	POSIX	Abort (kill)
11	* SIGSEGV	POSIX (HW)	Segmentation Violation
14	SIGALRM	POSIX	Alarm Clock
19	SIGSTOP		Stop
20	* SIGTSTP	GNAT	User stop requested from tty
21	* SIGTTIN	GNAT	Background tty read attempted
22	* SIGTTOU	GNAT	Background tty write attempted
26	SIGVTALRM		Virtual timer expired
27	* SIGPROF	GNAT	Profiling timer expired
31	SIGUNUSED		Unused signal

Signals marked with * are not allowed to be masked by the GNAT Run-Time. SIGINT can not be masked because it is used to terminate the Ada program when the CTRL-C sequence is pressed in the terminal that is controlling the process. By keeping SIGINT reserved, the programmer allows the user to do Ctrl-C but, in the same way, disable the ability of handling this signal in the Ada program.

GNAT Pragma *Unreserve_All_Interrupts* [Cor04] gives the programmer the ability to change this behavior. SIGILL, SIGFPE and SIGSEV can not be masked because they are used by the CPU to notify errors to the run-time. SIGTRAP is used by GNAT to enable debugging on multi-threaded applications. SIGABRT can not be masked because it is used by GNAT to implement the tasks abortion (described in chapter 20). SIGTTIN, SIGTTOU and SIGTSTP are not allowed to be masked so that background processes and IO behaves as normal C applications. Finally, SIGPROF can not be masked to avoid confusing the profiler.

19.2 Data Structures

No matter the association style used, GNARL always uses the following tables indexed by the *Interrupt_ID* to handle interrupts.

- *Table of Reserved Signals*: Booleans constant table[1] used to register reserved interrupts.

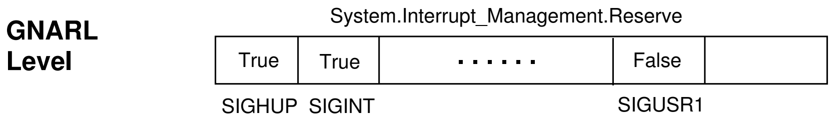


Figure 19.1: Reserved Interrupts Table.

- *User-defined Interrupt Handlers Table*: Table used to register and unregister the reference to *User-Defined Interrupt-Procedures* (UDIP) during the life of the program. Each element of this table is a record with two fields: the access to the UDIP and a flag which remembers the association style (nested or non-nested).

Figure 19.2 represents one protected procedure attached to signal SIGUSR1 in nested style (static style). The GNAT compiler associates two subprograms *P* and *N* to each protected subprogram (described in section 11.2.2). As the reader can see, the run-time links the signal with the *P* subprogram: the reference to the *P* subprogram is stored in the corresponding field of the table, and the *Static* field is set to *True* to remember that it is a nested style association.

[1]In the GNARL sources it is declared as variable just to be able to initialize it in the package body to aid portability.

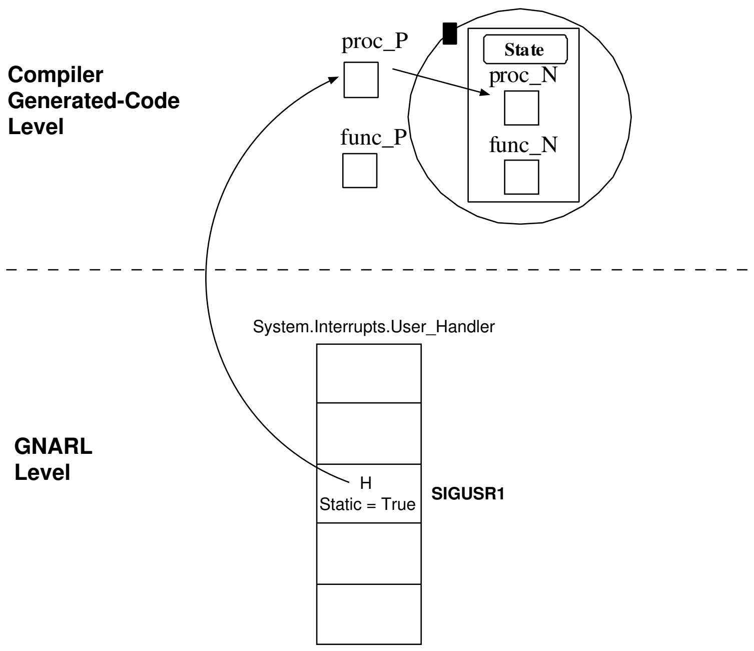


Figure 19.2: Table of User-Defined Interrupt-Handlers.

19.2.1 Interrupts Manager: Basic Approach

The GNAT run-time uses one *Interrupts Manager* task to serialize the execution of subprograms involved in the management of signals: attachment, detachment, replacement, etc. Figure 19.3 presents a simplified version of the automaton implemented by the Interrupt Manager. For simplicity we have considered only two basic operations: *Binding* and *Unbinding* User-Defined Interrrupt Procedures (UDIP) to interrupts.

First the automaton calls GNARL subprogram *Make_Independent* to do the *Interrupt Manager Task* independent of its masters. GNARL Independent tasks are associated with master 0, and their ATCBs are not registered in *All Tasks List* (described in section 14.1); thus they last until the end of the program. After the signal mask is set, the automaton goes to one state in which it waits for the next signal management operation.

- In case of signal *Binding*, GNARL saves the reference to the UDIP in its table, and blocks the POSIX signal (this allows GNARL to catch the signal with the *sigwait* POSIX service).
- In case of signal *Unbinding*, the reference to the UDIP is removed from the

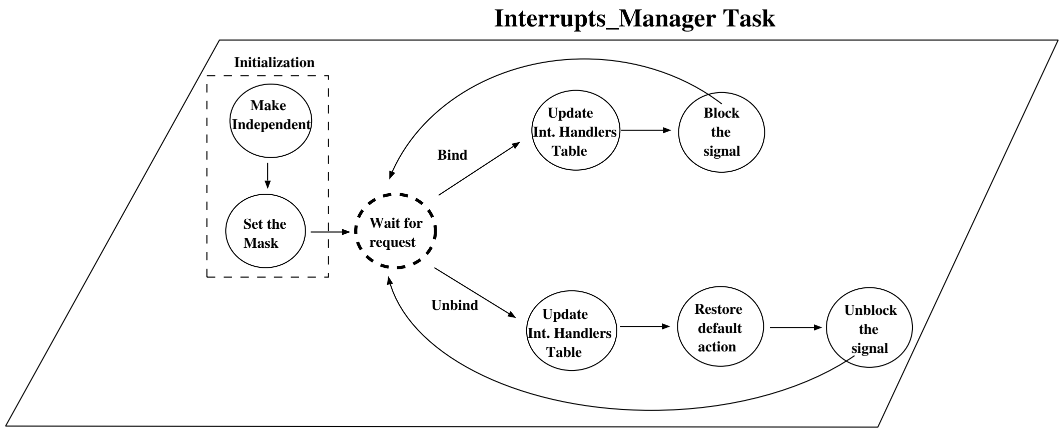


Figure 19.3: Basic Automaton Implemented by the *Interrupts Manager*.

table, the POSIX default action is set, and the signal is unblocked.

19.2.2 Server Tasks: Basic Approach

The Ada run-time must provide a thread to execute the UDIP. There is a choice between dedicating one server task for all signals and providing a server task for each signal. The former approach looks attractive, since it saves run-time space, but it blocks other signals during the protected procedure call. This may result in delayed or lost signals. For this reason, GNARL provides a separate *Server Task* for each signal [DIBM96].

Instead of create/abort *Server Tasks* when the user-defined interrupt handlers are attached/detached, GNARL keeps them alive until the program terminates. Thus they are reused by all UDIPs associated with the same interrupt during the life of the program. The run-time has a *Server_ID Table* which saves *Server Tasks* references (cf. Figure 19.4).

Figure 19.5 presents a simplified version of the Server Tasks Automaton.

19.2.3 Interrupt-Manager and Server-Tasks Integration

Previous sections have been concerned with the basic functionality of the *Interrupt Manager Task* and the *Server Tasks*. However, the GNARL implementation is a little more complex because:

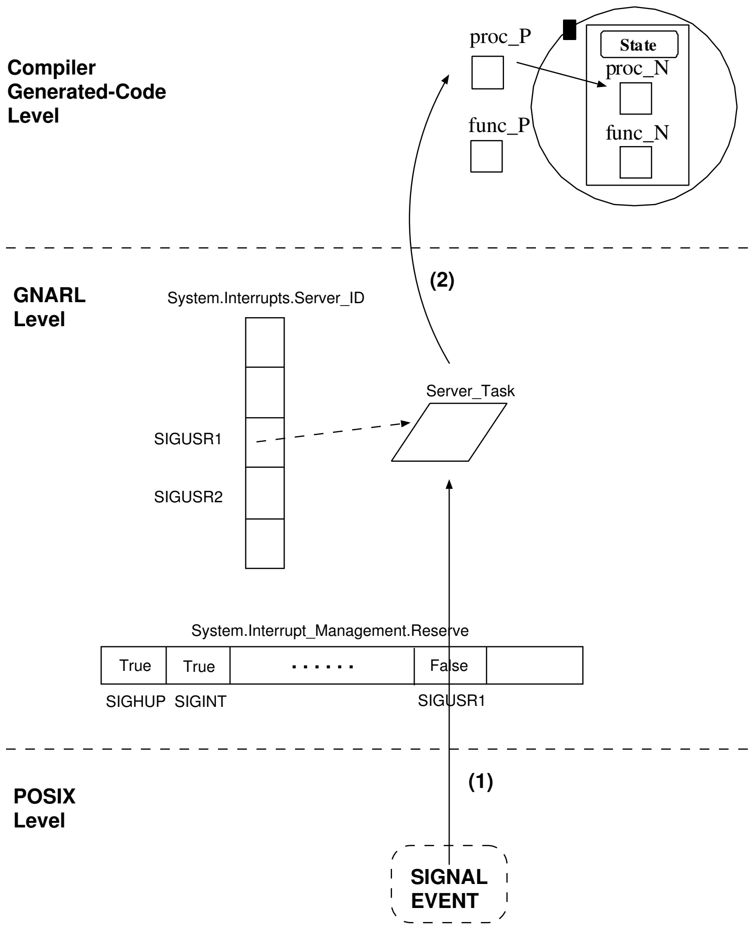


Figure 19.4: Server Tasks Signal Handling.

1. Ada nested style of interrupts implies that UDIPs are dynamically attached and detached to signals in the elaboration and finalization of protected objects. Therefore:

   (a) If no UDIP is registered GNARL must take the default POSIX action, and the simplified implementation of the *Interrupt Manager* did not consider POSIX default actions (cf. Figure 19.5).

   (b) When one UDIP is registered the signal is programmed to be handled by the UDIP. Following UDIPs registered to the same signal replace previous UDIPs.

   (c) If all UDIPs are detached, GNARL must again take the default POSIX action. The previous implementation can not achieve this effect so long as the *Server Task* is sitting on the *sigwait*. Even if the POSIX

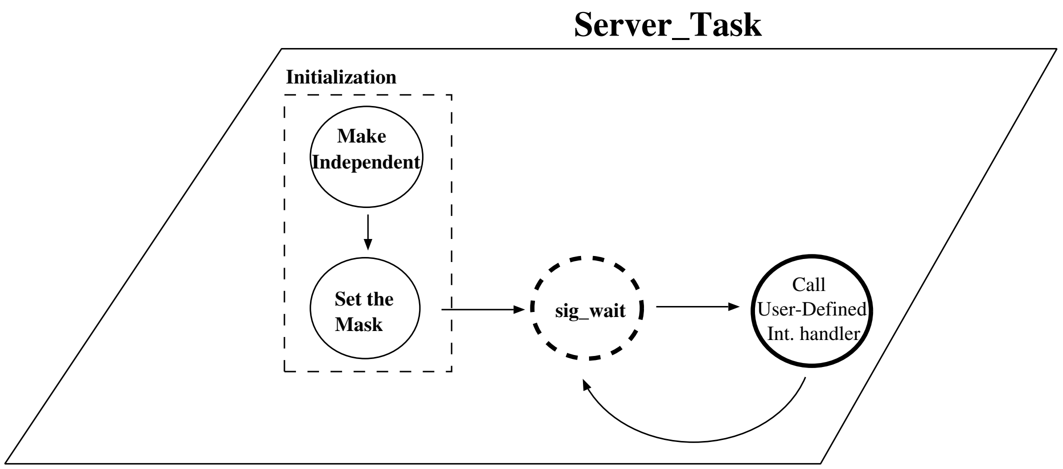


Figure 19.5: Basic Automaton Implemented by the Server Tasks.

sigaction command is used to set the asynchronous signal action to the default, that action will not be taken unless the signal is unmasked, and GNARL can not unmask the signal while the *Server Task* is blocked on *sigwait* because in POSIX.1c the effect is undefined. Therefore, GNARL must wake up the *Server Task* and cause it to wait on some operation instead for which it is safe to leave the signal unmasked, so that the default action can be taken [DIBM96].

2. GNARL must protect data structures shared by the *Interrupts Manager Task* and the *Server Tasks*. Therefore, some locks must be added.

The second requirement (*locks*) is easy to solve by means of POSIX mutexes. However, the first requirement is more complex. So let's focus our attention on the GNARL solution of the first requirement.

In order to better understand the GNARL implementation, we need to simplify the *Server Tasks Automaton* to its main states:

- *State 1*: The *Server Task* provides the POSIX default behavior of the signal.
- *State 2*: The *Server Task* has been programmed to call one UDIP.

In order to notify the automaton that it must jump from *State 1* to *State 2* GNARL uses one POSIX *Condition Variable*; in order to force the automaton to jump from *State 2* (waiting in the POSIX *sigwait* operation) to *State 1* the POSIX signal SIGABORT is used (this signal is used to kill the POSIX thread, and thus forces the *Server Task* to return from the POSIX *sigwait* operation). Figure 19.7 presents this automaton.

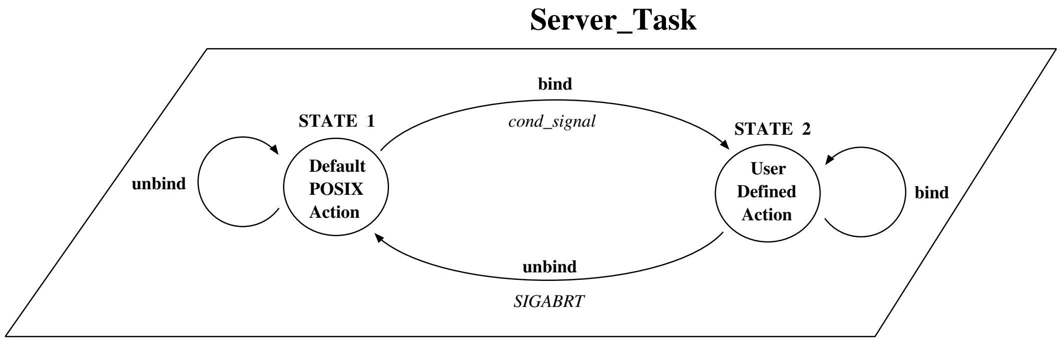


Figure 19.6: Simplified Server Tasks Automaton.

If we add these new transitions to our basic *Task Server Automaton* (cf. Figure 19.5) we have the real automaton implemented in GNARL (cf. Figure 19.7). In order to help the reading of the automaton all the states have been numbered. Inside dotted rectangles we find the states associated with the simplified states of the previous example (*State_1* and *State_2*).

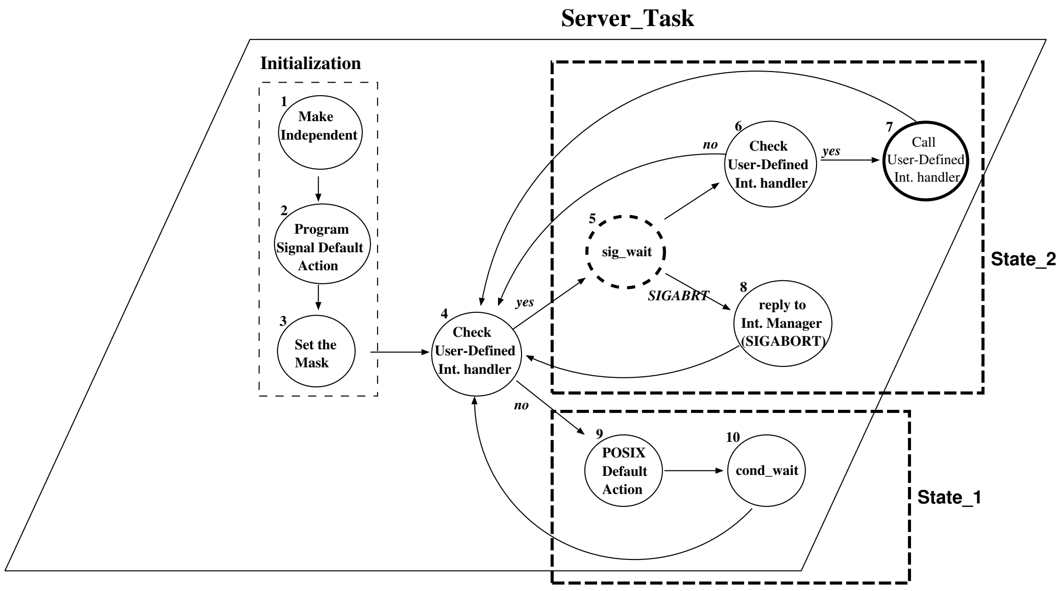

Server_Task
Initialization
1 Make Independent
2 Program Signal Default Action
3 Set the Mask
4 Check User-Defined Int. handler
5 sig_wait
6 Check User-Defined Int. handler
7 Call User-Defined Int. handler
8 reply to Int. Manager (SIGABORT)
9 POSIX Default Action
10 cond_wait
yes
no
yes
no
SIGABRT
State_2
State_1

Figure 19.7: Server Tasks Automaton.

After the initializations (states numbered 1 to 3), the automaton verifies if any UDIP has been registered by the *Interrupt Manager* (state 4). Initially, because no

UDIP has been registered, it takes the POSIX default action (state 9) and waits in the *Condition Variable* (*cond_wait*, state 10) until some UDIP is registered by the *Interrupt_Manager*.

When any UDIP is registered, the *Interrupt_Manager* signals the *Condition Variable* and the *Server Task Automaton* jumps to state 4, checks if some UDIP has been registered (now this evaluates to *True*) and jumps to state 5 to wait for the next signal occurrence. When the signal is received, it again checks if the UDIP is still registered (state 6), because it may have been removed by the *Interrupt Manager* while the automaton was waiting for the signal. Then it calls the UDIP (state 7) and again jumps to state 4.

While the *Server Task* is in state 5 waiting for the signal occurrence, it may happen that all UDIPs have been removed the *Interrupt Manager*. In this case the *Interrupt Manager* sends the SIGABRT signal to the *Server Task* to force it to jump to state 9. This signal wakes up the *Server Task Automaton*, which jumps to state 8 to reply to the *Interrupt Manager* with the same signal to inform it is not in state 5 (waiting for the signal). After this notification the automaton jumps to state 4 and, because no UDIP is found, it jumps to state 9.

19.3 Run-Time Subprograms

19.3.1 GNARL.Install_Handlers

In the nested style the expander generates a call to *Install_Handlers* in the initialization procedure of the protected object. This subprogram saves the previous handlers in one additional field of the object (*Previous_Handlers*) and installs the new handlers.

19.3.2 GNARL.Attach_Handlers

In the non-nested style, nothing special needs to be done since the default handlers will be restored as part of task completion which is done just before global finalization.

In order to verify at run-time that all the non-nested style interrupt procedures have been annotated with pragma *Interrupt_Handler* ([AAR95, Section C.3.2] requirement) the compiler adds calls to the GNARL subprogram *Register_Interrupt_Handler* to register these interrupt procedures in a GNARL single-linked list. The *Head* and *Tail* of this list are stored in two GNARL variables (*Registered_Handler_Head* and *System.Interrupts.Registered_Handler_Tail*, cf. Figure 19.8). Every node keeps the address of one protected procedure associated with an interrupt in non-nested style. For simplicity, a single access to a protected procedure has been represented; however, each node has the access to its corresponding *P* subprogram. Before the attachment of one non-nested style interrupt handler to one signal, GNARL traverses this list to verify that the protected procedure is registered in the list; otherwise it raises the exception *Program_Error*.

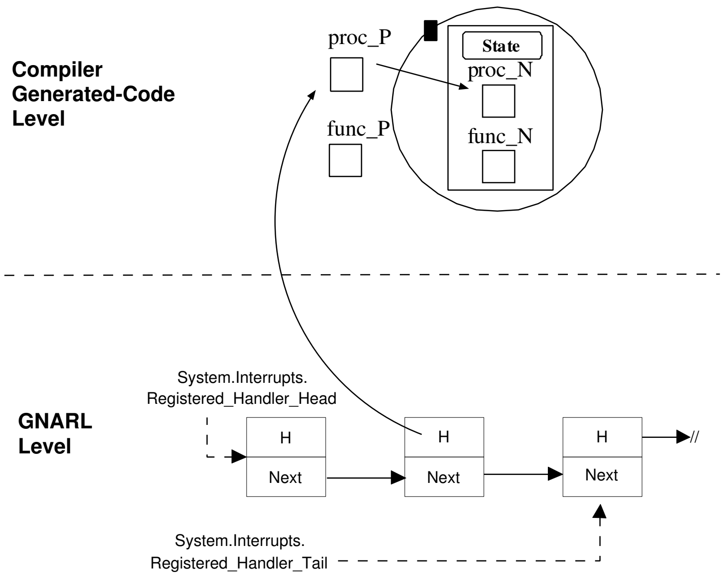

Compiler Generated-Code Level
proc_P
func_P
State
proc_N
func_N
GNARL Level
System.Interrupts. Registered_Handler_Head
H
Next
H
Next
H
Next
//
System.Interrupts. Registered_Handler_Tail

Figure 19.8: List of Interrupt Handlers in Non-Nested Style.

19.4 Summary

In this chapter we have dealt with the main aspects related to Interrupts Management. Although Ada allows us to attach a task entry to an interrupt, nowadays this is considered an obsolescent feature of the language. Thus, we have only discussed the attachment of User-Defined Protected-Procedures to interrupts. The main features of the GNAT implementation are:

- GNARL associates Ada interrupts to POSIX signals.
- Each signal has a *Task Wrapper* responsible for the execution of the User-Defined Protected-Procedures.

- The protected subprogram *P* associated with the protected subprogram is attached by GNARL to the corresponding *Task Wrapper*.
- Ada provides two ways to attach a protected procedure to an Ada interruption: nested style (by means of the pragma *Attach_Handler*) and non-nested style (by means of the pragma *Interrupt_Handler*).
  - When the nested style is used, GNARL adds one field to the run-time information of the protected object to save and restore the previous handler.
  - When the non-nested style is used, a dynamic link list is used to register non-nested style UDIPs. This list allows GNARL to verify that only non-nested UDIPs have been marked with the right pragma.
- An *Interrupt Manager Task* is used to serialize all the signal-management operations.

Chapter 20

Abortion

The abort statement is intended for use in response to those error conditions where recovery by the errant task is deemed to be impossible. Tasks which are aborted are said to become *abnormal*, and are thus prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately: the aborted task becomes *abnormal* and any non-completed tasks that depend upon an aborted task also become abnormal. Once all named tasks are marked as abnormal, then the abort statement is complete, and the task executing the abort can continue. It does not wait until named tasks have actually terminated [BW98, Section 10.2].

After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted. However, certain actions must be protected in order that the integrity of the remaining tasks and their data be assured. The following operations are defined to be abort-deferred []: a protected action, waiting for an entry call to complete, waiting for termination of dependent tasks, and the execution of an "initialize" procedure, a "finalize" procedure, or an assignment operation of an object with a controlled part. In addition, the run-time also needs to defer abortion during the execution of some run-time subprograms to ensure the integrity of its data structures. The language also defines the *abort-completion points*[]: 1) The end of activation of a task, 2) The point where the execution initiates the activation of another task, 3) The start or end of an entry call, accept statement, delay statement or abort statement, and 4) The start of the execution of a select statement, or of the sequence or statements of an exception handler.

In general, processing an abort requires unwinding the stack of the target task, rather than immediately jumping out of the aborted part (or killing the task, in the case of entire-task abortion). There may be local controlled objects, which require the execution of a finalization routine. There also may be dependent tasks, which require the aborted processing block until they have been aborted, finalized, and terminated. The finalization must be done in LIFO order and the stack contexts of the objects requiring finalization must be preserved until the objects are finalized [GB94, Section 3.4]

Abort-deferral implementation can be divided into two parts [GB94, Section 3.3]: 1) determine whether abort is deferred for a given task, at the point it is targeted for abortion, and 2) ensure deferred aborts are processed immediately when abort-deferral is lifted. In general, the determination of whether a given task is abort-deferred must be carried out by the task itself. In a single-processor system, it may be possible for the task initiating an abort to determine whether the target task is abort-deferred. However, in a multi-processor system, or a single processor system where the Ada run-time is not in direct control of task scheduling, this is not possible. The abort-deferral state of the target task may change between the point it is tested and the point the target task is interrupted.

There are two obvious techniques for recording whether a task is abort-deferred. One technique is sometimes termed PCmapping. The compiler and link-editor generate a map of abort-deferred regions. Whether the task is abort-deferred can then be determined by comparing the current instruction-pointer value, and all the saved return addresses of active subprogram calls, against the map. To ensure the abort is processed on exit from the abort-deferred region, one overwrites the saved return address of the outermost abort-deferred call frame with the address of the abort-processing routine (saving the old return address elsewhere). The test for abort deferral may take time proportional to the depth of subprogram call nesting, but that occurs only if an ATC is attempted. Until that occurs, no runtime overhead is incurred for abort deferral. A restriction of this method is that abort-deferred regions must correspond to callable units of code. Another restriction is that the subprogram calling convention is constrained to (1) ensure the return addresses are always in a predictable and accessible location and (2) ensure this data is always valid, even if the calling sequence is interrupted. Unfortunately, that is not true for some architectures [GB94, Section 3.3].

In the second technique the task increments and decrements a deferral nesting level (e.g. in a dedicated register or the ATCB), whenever it enters and exits an abort-deferred region. On exit from such a region, if the counter goes to zero, the task must check whether there is a pending abort and, if so, process the abort. This deferral-counter method imposes some distributed overhead on entry and exit of abort-deferred regions, but allows quick checking [GB94, Section 3.3]. The GNAT run-time implements this second technique.

ATC implementation must address the following issues [GB94, Section 3]:

- Interruption of the target task and abortion initiation.
- Deferral of abort over certain regions.
- Execution of finalization procedures for any local objects in the aborted part, each in its correct context.
- Finding the proper location and context to continue execution, after the ATC.
- Handling nested scopes, including nested asynchronous select statements.
- Ensuring safety of compiler-generated code sequences, including subprogram call and return when interrupted by ATC.

ATC is very much like exception propagation, so it is desirable that one mechanism serve for both purposes. Since ATC is not likely to be used in non real-time Ada programs, a key objective of any implementation should be to impose little or no distributed overhead for the existence of this language feature. In principle, some efficiency might be gained by avoiding detailed unwinding of the stack, executing the finalization routines from the top of the stack or from a different stack, then poping the entire stack down to the context where control is to be transferred. However, this presumes there is some way to recover that context without full unwinding. If the compiler uses a callee-save register spilling convention, there may be values of live registers spilled at unpredictable locations on the stack. In this case, it seems one must create a register save area for each potential target of ATC (analogous to jump-buffer implementation of C's setjmp() and longjmp() operations). While asynchronous select statements may not be very common, exception handlers are common (some implicitly provided by the compiler), and controlled objects are also expected to be common. Thus, the overhead of creating a jump-buffer for every potential asynchronous transfer point is objectionable [GB94, Section 3.4].

Some means must be provided for locating finalization routines, and the point at which execution is to resume after an ATC. This problem is very similar to that of finding an exception handler, and the same solutions apply. The main approaches are saving a pointer in the stack frame for each scope, PC-mapping, and various hybrids of the two. The PC-mapping approach is generally preferable, since it imposes no distributed overhead on execution. If PC-mapping is used for the latter purpose, there is strong motivation to try to make the same technique serve double duty, for abort-deferral [GB94, Section 3.4].

20.1 Run-Time Subprograms

The GNAT implementation of abortion is made up of:

- One flag in the ATCB (*Aborting*). This flag prevents a race between multiple aborters and the aborted task. This is essential since an aborter may be preempted and would send the abortion signal when resuming execution.
- One internal exception (*_Abort_Signal*). This exception is not visible to user code (cf. Section 18.1.1); it can only be caught by run-time code. Its propagation performs finalization of all the scopes along the way.
- One POSIX signal (SIGABRT), which can not be masked.

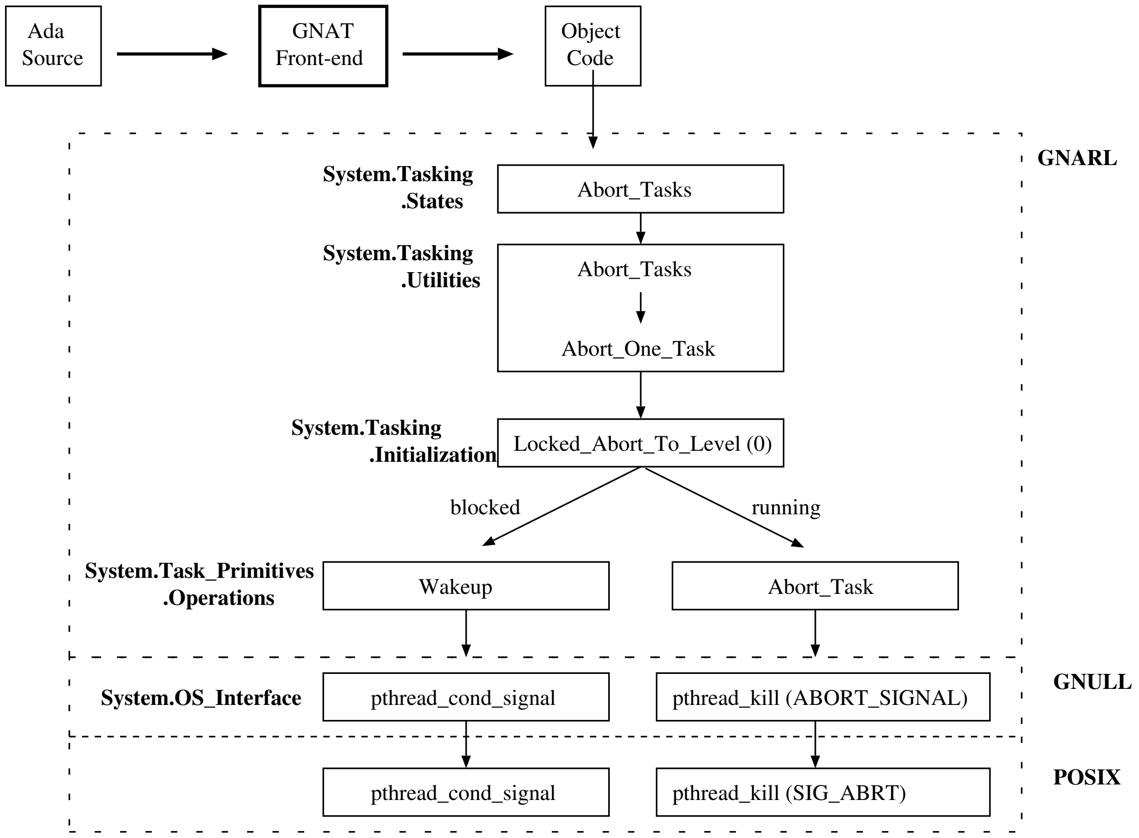


Figure 20.1: GNARL Subprograms for the Abort Statement.

Figure 20.1 presents the sequence of run-time subprograms involved in the task abortion, which are described in the following sections.

20.1.1 GNARL.Task_Entry_Call

The GNAT run-time subprogram *Task_Entry_Call* (cf. Section 15.5.4) not only gives support to normal entry-calls but also the ATC entry-calls. In this latter case, because ATCs can be nested, the run-time needs to store all these pending entry-calls. For this purpose, the GNAT run-time associates an *entry-call stack* to each Ada task (cf. Figure 20.2). The *Pending_ATC_Level* ATCB field is used to signal an ATC abortion. In order to distinguish the *Abort* statement from the ATC abortion, the run-time defines the following rules:

- In the case of an **abort** statement, *Pending_ATC_Level* is set to 0.
- In the case of an ATC abortion, *Pending_ATC_Level* is set to the level in which the caller was just before the entry call was made (*ATC_Nesting_Level* minus one).

20.1.2 GNARL.Locked_Abort_To_Level

GNARL.Undefer_Abort subprogram is the universal polling point for deferred processing. It gives support to base priority changes, exception handling, and asynchronous transfer of control (ATC). In case of base-priority change, after the new priority is set, it yields the processor so that the scheduler chooses the next tasks to execute. In the other cases, it verifies if there is some pending exception to raise (ATC abortion raises the internal exception *Abort_Signal*).

20.1.3 GNARL.Locked_Abort_To_Level

Locked_Abort_To_Level sets to true the ATCB flag *Pending_Action.* and, depending on the current state of the target task (blocked or running) it calls *GNARL.Wakeup* or *GNARL.Abort_Task*:

- If the task to be aborted is in a *sleep* state (cf. Section 14.2), it is in a deferred abortion section. Therefore, when in the future, the aborted task is woken up and continues its execution, it executes the Undefer_Abortion subprogram. At this moment the Pending_Action ATCB flag will be checked and, being true, it sets the ATCB flag *Aborting* and raises the internal exception *_Abort_Signal.*

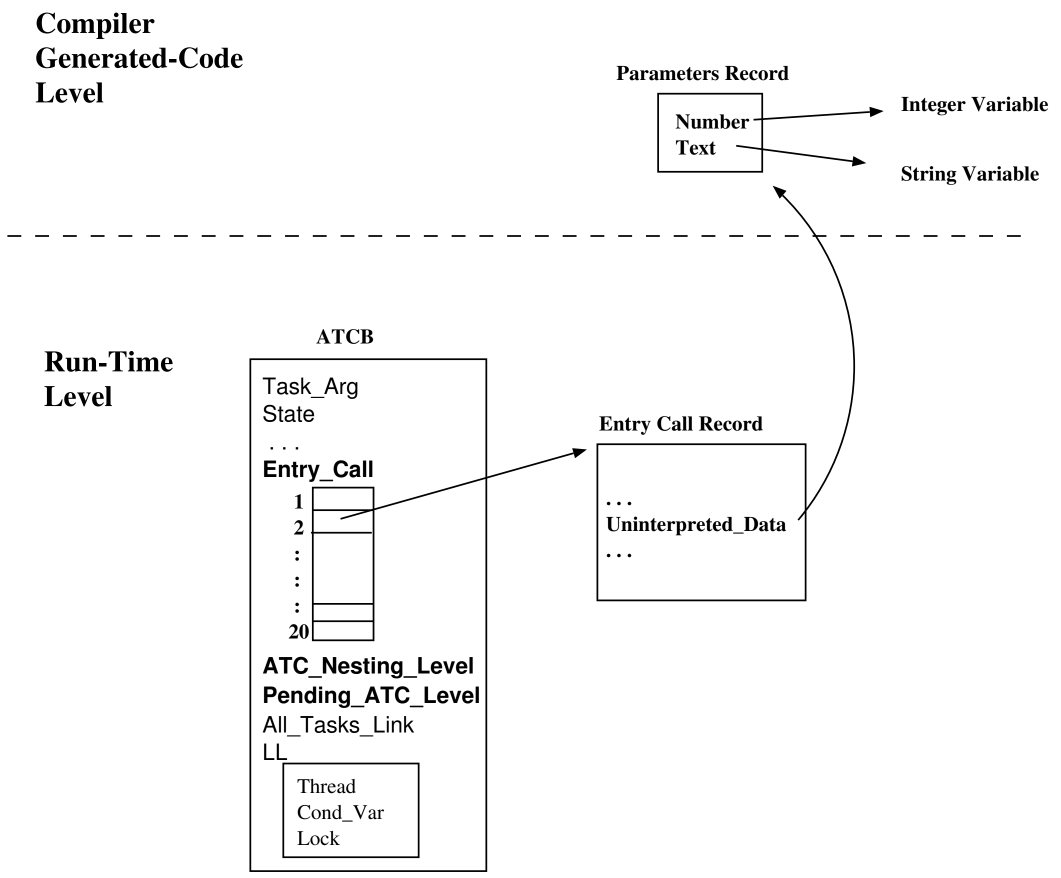


Figure 20.2: Entry Calls Stack.

- If the task is in the *running* state then the aborter sends to it the signal *SIG_ABRT* and then the corresponding run-time *abort-handler* asynchronously raises the internal exception *_Abort_Signal* in the aborted task.

In both cases the internal exception *Abort_Signal* unwinds the stack of the aborted task.

20.2 Summary

The GNARL implementation of the Ada abort statement is made up of one flag in the ATCB: *Aborting*, one exception _Abort_Signal, and one POSIX signal (SIGABRT). The flag prevents a race between multiple aborters and the aborted task.

The exception is can only be handled by run-time system code. The POSIX signal can not be masked.

Part V

Fifth Part: Appendix

Appendix A

How to add new Keywords, Pragmas and Attributes

This appendix describes how to modify the GNAT front-end to experiment with Ada extensions. As an example we use **Drago**, an experimental extension of Ada 83 designed to support the implementation of fault-tolerant distributed applications. It was the result of an effort to impose discipline and give linguistic support to the group communication paradigm. In the following sections we briefly introduce Drago (Section A.1), and describe the modifications made to the GNAT scanner, parser, and semantic analyzer. A previous version of this work was publised in [MGMG99].

A.1 Drago

Drago [MAAG96, MAGA00] is an experimental language developed as an extension of Ada for the construction of fault-tolerant distributed applications. The hardware assumptions are: a distributed system with no memory shared among the different nodes, a reliable communication network with no partitions, and *fail-silent* nodes (that is, nodes which once failed are never heard from again by the rest of the system.) The language is the result of an effort to impose discipline and give linguistic support to the main concepts of Isis [BCJ$^+$90], as well as to experiment with the group communication paradigm. To help build fault-tolerant distributed applications, Drago explicitly supports two process group paradigms, *replicated process groups* and *cooperative process groups*. Replicated process groups allow the programming of fault-tolerant applications according to the active replication model, while cooperative process groups permit programmers to express parallelism and therefore increase throughput.

A process group in Drago is actually a collection of *agents*, which is the way processes are identified in the language. Agents are rather similar in appearance to Ada tasks (they have an internal state not directly accessible from outside the agent, an independent flow of control, and public operations called *entries*). Furthermore, they are the unit of distribution in Drago and in this sense they perform a role similar to Ada 95 active partitions and Ada 83 programs. Each agent resides in a single node of the network, although several agents can reside in the same node. A Drago program is composed of a number of agents residing at a number of nodes.

A.2 First Step: Addition of new Keywords

Drago adds four reserved keywords (**agent**, **group**, **intragroup**, and **replicated**). In the following sections we describe the main steps required to introduce them into the GNAT environment. GNAT list of predefined identifiers contains all the supported pragmas, attributes and keywords. This list is declared in the specification of package *Snames*. For each predefined identifier there is a constant declaration which records its position in the *Names Table*. This hash table stores all the names, predefined or not. Keywords are classified in two main groups: keywords shared by Ada 83 and Ada 95, and exclusive Ada 95 keywords. Each group is delimited by means of a subtype declaration. Depending on the GNAT compilation mode, Ada 83 or Ada 95, this subtype allows the scanner to properly distinguish user identifiers from Ada keywords.

In order to introduce Drago keywords we added a third GNAT mode, Drago mode, and one new group with Drago exclusive keywords. The result was as follows:

```
First_Drago_Reserved_Word  : constant Name_Id := N + 475;
Name_Agent                 : constant Name_Id := N + 475;
Name_Group                 : constant Name_Id := N + 476;
Name_Intragroup            : constant Name_Id := N + 477;
Name_Replicated            : constant Name_Id := N + 478;
Last_Drago_Reserved_Word   : constant Name_Id := N + 478;

subtype Drago_Reserved_Words is
   Name_Id range First_Drago_Reserved_Word
                .. Last_Drago_Reserved_Word;
```

We also updated the value of the constant *Preset_Names*, declared in the body of *Snames*, keeping the order specified in the previous declarations. This constant contains the literals of all the predefined identifiers.

A.3 Second step: Addition of new tokens

The list of tokens is declared in the package *Scans*. It is an enumerated type whose elements are grouped into classes used for source tests by the parser. For example, *Eterm* class contains all the expression terminators; *Sterm* class contains the simple expressions terminators[1]; *After_SM* is the class of tokens that can appear after a semicolon; *Declk* is the class of keywords which start a declaration; *Deckn* is the class of keywords which start a declaration but can not start a compilation unit; and *Cunit* is the class of tokens which can begin a compilation unit. Members of each class are alphabetically ordered. We have introduced the new tokens in the following way:

```
type Token_Type is (
   --  Token name        Class(es)
   ...
   Tok_Intragroup,    -- Eterm, Sterm, After_SM
   ...
   Tok_Agent,         -- Eterm, Sterm, Cunit, Declk, After_SM
   ...
   Tok_Group,         -- Eterm, Sterm, Cunit, Declk, After_SM
   ...
   Tok_Replicated,    -- Eterm, Sterm, Cunit, After_SM
   ...
   No_Token);
```

Classes associated with tokens are specified in the third column. Our choices were based on the following guidelines:

- **Intragroup** must always appear after a semicolon (see th specification of a Drago group on section A.6).
- **Agent** and **Group** start a compilation unit and a new declaration.

[1]All the reserved keywords, except *mod, rem, new, abs, others, null, delta, digits, range, and, or xor, in* and *not*, are always members of these two classes (*Eterm, Sterm*).

- **Replicated** qualifies a group (similar to Ada 95 private packages, where the word *private* preceding a package declaration qualifies the package; they are otherwise public). Therefore they were placed in the same section.

According to the alphabetic ordering, *Tok_Agent* is new first token of *Cunit* class. Therefore we updated the declaration of the corresponding subtype *Tok_-Class_Unit* to start the class with *Tok_Agent*. Finally we modified the declaration of the table *Is_Reserved_Keyword*, which records which tokens are reserved keywords of the language.

A.4 Third Step: Update the Scanner Initialization

The scanner initialization (subprogram *Scn.Initialization*) is responsible for stamping all the keywords stored in the *Names Table* with the byte code of their corresponding token (0 otherwise). This allows the scanner to determine if a word is an identifier of a reserved keyword. This work is done by means of repeated calls to the procedure *Set_Name_Table_Byte* passing the keyword and its corresponding token byte as parameters. Therefore we added the following sentences to the scanner initialization:

```
...
Set_Name_Table_Byte (Name_Agent,
                     Token_Type'Pos (Tok_Agent));
Set_Name_Table_Byte (Name_Group,
                     Token_Type'Pos (Tok_Group));
Set_Name_Table_Byte (Name_Intragroup,
                     Token_Type'Pos (Tok_Intragroup));
Set_Name_Table_Byte (Name_Replicated,
                     Token_Type'Pos (Tok_Replicated));
```

We also modified the scanner (subprogram *Scn.Scan*) in order to recognize the new keywords only when compiling a Drago program. This allows us to preserve its original behaviour when analyzing Ada source code. This was the last modification required to integrate the new keywords into GNAT. In the following section we describe the modifications made to add one new pragma and one attribute into the GNAT scanner.

A.5 Addition of Pragmas an Attributes

Drago provides one new attribute *Member_Identifier* and one new pragma (*Drago*). When *Member_Identifier* is applied to a group identifier it returns the identifier of the current agent in the specified group. When pragma *Drago* is applied the compiler is notified about the existence of Drago code (similar to GNAT pragmas *Ada83* and *Ada95*). For integrating them into GNAT we had to modify the package *Snames* in the following way:

1. Add their declaration to the list of predefined identifiers keeping the alphabetic order. GNAT classifies all pragmas in two groups: configuration pragmas, those used to select a partition-wide or system-wide option, and non-configuration pragmas. The pragma Drago was placed in the group of non-configuration pragmas.

   GNAT classifies all attributes in four groups: attributes that designate procedures (*output, read* and *write*), attributes that return entities (*elab_body* and *elab_spec*), attributes that return types (*base* and *class*), and the rest of the attributes. *Member_Identifier* was placed in this fourth group.

2. Insert their declarations in the enumerated *Pragma_Id* and *Attribute_Id* keeping the order specified in the previous step. Similar to tokens associated with keywords, these types facilitate the handling of pragmas and attributes in later stages of the frontend.

3. Add their literals in *Preset_Names*. Similar to the introduction of the keywords, we must keep the order specified in the list of predefined identifiers.

4. Update the C file *a-snames.h*. This file associates a C macro to each element of the types *Attribute_Id* and *Pragma_Id*. This work can be automatically done by means of the GNAT utility *xsnames*.

A.6 Addition of New Syntax Rules

In this section we describe, by means of an example, the modifications made in the parser in order to support Drago syntax. The example is the specification of a Drago group, whose syntax is similar to the one of an Ada package specification:

```
GROUP_DECLARATION ::= GROUP_SPECIFICATION

GROUP_SPECIFICATION ::=
  [ replicated ] group defining_identifier is
     {basic_declarative_item}
  [ intragroup
     {basic_declarative_item} ]
  [ private
     {basic_declarative_item ]
  end [ group_identifier];
```

Replicated groups are denoted by the reserved keyword **replicated** at the heading of the group specification. Cooperative groups do not require any reserved word because they are considered the default group specification. The first list of declarative items of a group specification (the *intergroup section*) contains all the information that clients are able to know about this group. The optional list of declarative items after the keyword **intragroup** is called the *intragroup section*. It contains information that only members of the group are able to know, and it can be declared only in a cooperative group specification[2]. The optional list of declarative items after the reserved word **private** is called the *private section* and provides groups with the same functionality as the private part of Ada packages. The following sections describe the steps made in order to add this syntax to the GNAT parser.

A.6.1 First step: Addition of New Node Kinds

GNAT node kinds are declared in the enumerated *Sinfo.Node_Kind*. Similar to *Token_Type* elements, all its elements are grouped into classes (i.e. nodes that correspond to sentences, nodes which correspond to operators, ...), and elements of each class are alphabetically ordered.

The addition of the rules of a Drago group required two additional kinds of nodes: *N_Group_Declaration* and *N_Group_Specification*. Due to the similarity of a Drago group specification and an Ada package specification we placed the *N_Group_Declaration* node in the class associated with *N_Package_Declaration* node, and *N_Group_Specification* in the class associated with *N_Package_Specification*.

[2]Replicated groups do not have this facility because their members are assumed to be replicas of a deterministic automaton and thus they do not need to exchange their state —all the replicas have the same state.

A.6.2 Second Step: High-level specification of the new nodes

The specification of package *Sinfo* contains the high level specification of the AST nodes (cf. Section 2.2.1). When we define a new node we have two possibilities: to reuse the AST field-names used in the current high-level specification of Ada, or to define new names. In the first case we must keep all its features: field-number and associated data. In the second case we must carefully analyze the field to which we associate the new names because once it is stated it must be kept fixed for all nodes; in addition, for each new name we must declare two subprograms in *Sinfo*: one procedure (used to set the value of the field), and one function (to get the stored value). Following with out example, the templates associated with *N_Group_Declaration* and *N_Group_Specification* are:

```
-- N_Group_Declaration
-- Sloc points to GROUP
-- Specification (Node1)

-- N_Group_Specification
-- Sloc points to GROUP
-- Defining_Identifier (Node1)
-- Visible_Declarations (List2)
-- Intragroup_Declarations (List3) (set to No_List if
--   no intragroup part present)
-- Private_Declarations (List4) (set to No\_List if
--   no private part present)
```

This means that the value of *Sloc* in a *N_Group_Declaration* node points to the source code word **group**, and the first field of the node (*Field1*) points to a specification node. On the other hand, the value of *Sloc* in a *N_Group_Specification* node also points to the same word **group** (because the reason for the creation of both nodes was the same word), its first field (*Field1*) points to a defining identifier node, and its second, third and fourth fields (*Field2..Field4*) point to lists which contain respectively its visible, intragroup and private declarations.

Similar to GNAT handling of private packages, the handling of replicated groups only requires the addition of one new flag in the AST root node (*Flag15* for a private package, and we chose *Flag16* for a replicated group).

A.6.3 Third Step: Automatic modification of the frontend

The templates specified in the previous step are used by three GNAT utility programs (*xsinfo, xtreeprs* and *xnmake*) to automatically generate four frontend source files involved in the handling of nodes: *a-sinfo.h, treeprs.ads, nmake.ads* and *nmake.adb* (cf. Figure A.1).

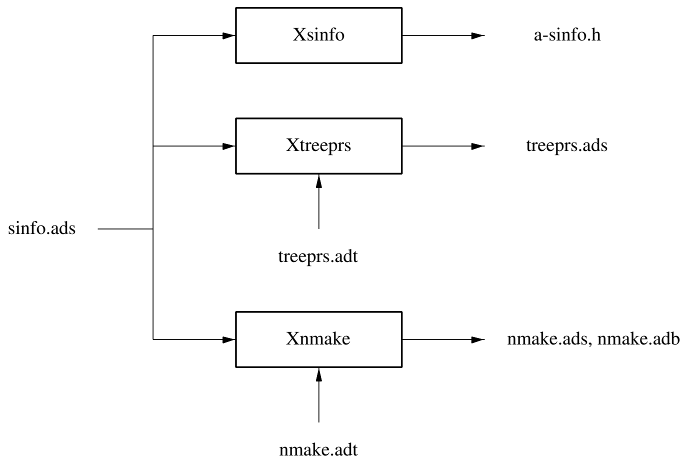


Figure A.1: GNAT utility programs.

A.6.4 Fourth Step: Update of the Parser

The GNAT parser is implemented by means of the well known recursive descent technique. All its code is inside the function *Par* which is composed of several subunits (one subunit for each Ada Reference Manual Chapter [AAR95]). According to this philosophy we decided to add the subunit *Par.Drag* to group all the parser code that syntactically analyzes Drago rules (figure A.2). We gave the name *P_Group* to the function associated with the parsing of a group specification. We used the fragment of the parser that analyzes a package specification as a reference for its development. Finally we modified the parsing of a compilation unit to include a call to this function when analyzing Drago source code and it detects a group token.

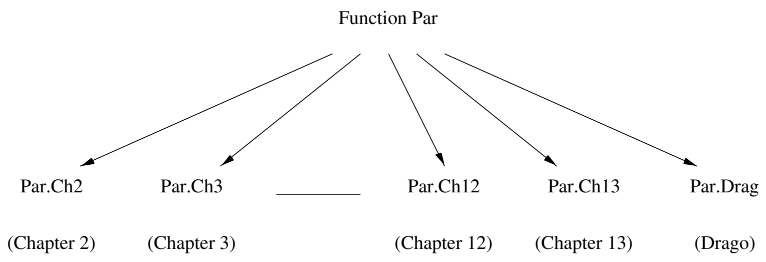


Figure A.2: New parser structure.

A.7 Verification of the Semantics

When the parser finishes its work, the semantic analizer makes a top-down traversal of the AST and, according to the kind of each visited node, it calls one subprogram that semantically verifies it. These subprograms are structured as a set of packages. Each package contains all the subprograms required to verify the semantic rules of each ARM chapter (packages *Sem_Ch2..Sem_Ch13*). There is one subprogram for each node type created by the parser and the main package that derivates the calls is called *Sem*.

For the semantic verification of a Drago group specification we made the following modifications to the GNAT sources:

1. Add one new package: *Sem_Drag*. This package contains all the Drago subprograms that make the semantic verifications.

2. Add one new semantic entity. A GNAT entity is an internal representation of the identifiers, operators and character literals found in the source program declarations. Therefore, the identifier of a group specification must have its corresponding entity. We added the element *E_Group* to the enumerated *Entity_Info* (inside the *Einfo* package) to represent the new group name entity.

3. Update the C file *a-einfo.h*. This work is automatically done by means of the GNAT utility program names *xeinfo* (figure A.3).

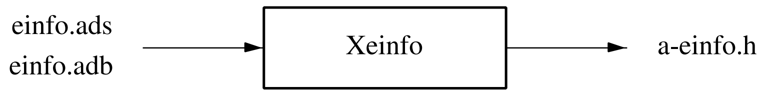


Figure A.3: GNAT semantic utility program

4. Write one subprogram for each new kind of node declared in the parser. We wrote the subprogram associated with the group specification node, and the subprogram for the group declaration node, and placed them inside the new package *Sem_Drag*. We used the subprograms performing semantic analysis of a package specification as a model for our new subprograms.

Finally we modified the *Sem* package to include calls to these subprograms when analyzing a group specification (or declaration) node.

A.8 Tree expansion

In the general case, to generate code there is no need to access the GIGI level, and our work finishes at the expansion phase. Following with our example, we expanded the Drago nodes into Ada 95 nodes which called several added Run-Time subprograms. Again, we recommend to use the expansion subprograms available in the GNAT sources as a reference.

A.9 Summary

This appendix has described the integration of Drago into the GNAT frontend. Drago is a language designed as an Ada extension to experiment with the active replication model of fault-tolerant distributed programming. We have focused our attention on the lexical, syntactic and semantic aspects of the integration. The abstract syntax tree expansion and code generation have not been discussed here.

Appendix B

Glossary

Access type. An access type has values that designate aliased objects. Access types correspond to “pointer types” or “reference types” in some other languages.

Aliased. An aliased view of an object is one that can be designated by an access value. Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word aliased. The Access attribute can be used to create an access value designating an aliased object.

Class. A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations.

Compilation unit. The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit.

Compile-time rules. Rules that are enforced at compile type.

Complete context. Language construct over which overload resolution must be performed without examining a larger portion of the program. Each of the following constructs is a complete context: a context item, a declarative_item or declaration, a statement, a pragma_argument_association, and the expresion of a case_statement [AAR95, Section 8.6(4-9)].

Composite type. A composite type has components.

Construct. A construct is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax."

Controlled type. A controlled type supports user-defined assignment and finalization. Objects are always finalized before being destroyed.

Declaration. A declaration is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an explicit declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).

Definition (view) All declarations contain a definition for a view of an entity. A view consists of an identification of the entity (the entity of the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see ARM 8.5)).

Derived type. A derived type is a type defined in terms of another type, which is the parent type of the derived type. Each class containing the parent type also contains the derived type. The derived type inherits properties such as components and primitive operations from the parent. A type together with the types derived from it (directly or indirectly) form a derivation class.

Discrete type. A discrete type is either an integer type or an enumeration type. Discrete types may be used, for example, in case_statements and as array indices.

Discriminant. A discriminant is a parameter of a composite type. It can control, for example, the bounds of a component of the type if that type is an array type. A discriminant of a task type can be used to pass data to a task of the type upon creation.

Dynamic semantics (see run-time semantics).

Elementary type. An elementary type does not have components.

Enumeration type. An enumeration type is defined by an enumeration of its values, which may be named by identifiers or character literals.

Exception. An exception represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an exception occurrence. To raise an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called handling the exception.

Execution. The process by which a construct achieves its run-time effect is called execution. Execution of a declaration is also called elaboration. Execution of an expression is also called evaluation.

Expanded Name. A selected_component is called an expanded name if according to the visibility rules, at least one possible interpretation of its prefix denotes a package or an enclosing named construct (directly, not through a subprogram_renaming_declaration or generic_renaming_declaration) [AAR95, Section 4.1.3(4)].

Generic unit. A generic unit is a template for a (nongeneric) program unit; the template can be parameterized by objects, types, subprograms, and packages. An instance of a generic unit is created by a generic_instantiation. The rules of the language are enforced when a generic unit is compiled, using a generic contract model; additional checks are performed upon instantiation to verify the contract is met. That is, the declaration of a generic unit represents a contract between the body of the generic and instances of the generic. Generic units can be used to perform the role that macros sometimes play in other languages.

Integer type. Integer types comprise the signed integer types and the modular types. A signed integer type has a base range that includes both positive and negative numbers, and has operations that may raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with "wraparound" semantics. Modular types subsume what are called "unsigned types" in some other languages.

Legality rules (see Compile-time rules).

Library unit. A library unit is a separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a subsystem.

Limited type. A limited type is (a view of) a type for which the assignment operation is not allowed. A nonlimited type is a (view of a) type for which the assignment operation is allowed.

Name resolution. Process that establishes a mapping between names and the defining entity referred to by the names at each point in the program. In the context of the GNAT semantic analyzer, name resolution involves to link each node that denotes an entity with its corresponding defining-entity node.

Name resolution rules. Compile-time rules used in name resolution, including overload resolution.

Object. An object is either a constant or a variable. An object contains a value. An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

Package. Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

Pragma. A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas.

Primitive operations. The primitive operations of a type are the operations (such as subprograms) declared together with the type declaration. They are inherited by other types in the same class of types. For a tagged type, the primitive subprograms are dispatching subprograms, providing run-time polymorphism. A dispatching subprogram may be called with statically tagged operands, in which case the subprogram body invoked is determined at compile time. Alternatively, a dispatching subprogram may be called using a dispatching call, in which case the subprogram body invoked is determined at run time.

Private extension. A private extension is like a record extension, except that the components of the extension part are hidden from its clients.

Private type. A private type is a partial view of a type whose full view is hidden from its clients.

Program unit. A program unit is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

Protected type. A protected type is a composite type whose components are protected from concurrent access by multiple tasks.

Record extension. A record extension is a type that extends another type by adding additional components.

Record type. A record type is a composite type consisting of zero or more named components, possibly of different types.

Run-time semantics (dynamic semantics). Definition of the run-time effect of each construct.

Scalar type. A scalar type is either a discrete type or a real type.

Subtype. A subtype is a type together with a constraint, which constrains the values of the subtype to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

Tagged type. The objects of a tagged type have a run-time type tag, which indicates the specific type with which the object was originally created. An operand of a class-wide tagged type can be used in a dispatching call; the tag indicates which subprogram body to invoke. Nondispatching calls, in which the subprogram body to invoke is determined at compile time, are also allowed. Tagged types may be extended with additional components.

Task type. A task type is a composite type whose values are tasks, which are active entities that may execute concurrently with other tasks. The top-level task of a partition is called the environment task.

Type. Each object has a type. A type has an associated set of values, and a set of primitive operations which implement the fundamental aspects of its semantics. Types are grouped into classes. The types of a given class share a set of primitive operations. Classes are closed under derivation; that is, if a type is in a class, then all of its derivatives are in that class.

View. (See Definition.)

Appendix C

GNU Free Documentation License

Preamble

subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some

2. VERBATIM COPYING

the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ”History” in the various original documents, forming one section Entitled ”History”; likewise combine any sections Entitled ”Acknowledgements”, and any sections Entitled ”Dedications”. You must delete all sections Entitled ”Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[AAR95] AARM. *Annoted Ada Reference Manual (Technical Corrigendum 1).* ISO/IEC 8652:1995(E), 1995.

[Bar95] Barnes, J. (ed.). *Ada 95 Rationale: The language. The Standard libraries.* Springer, 1995.

[Bar99] J. Barnes. *Programming in Ada 95 (2nd edition).* Addison-Wesley, 1999.

[BCJ$^+$90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual. Version 2.1.* Cornell University, Cornell, September 1990.

[BR85] T.P. Baker and G.A. Riccardi. Ada Tasking: from Semantics to Efficient Implementation. *Florida-State University*, 1985.

[BW98] A. Burns and A. Wellings. *Concurrency in Ada (2nd edition).* Cambridge University Press, 1998.

[CDG95] C. Comar, G. Dismukes, and F. Gasperoni. The GNAT Implementation of Controlled Types. *Proceedings of Tri-Ada'95*, pages 467–473, 1995.

[CGS94] C. Comar, F. Gasperoni, and E. Schonberg. The GNAT Project: A GNU-Ada9X Compiler. Technical report, New York University, 1994.

[Coh96] N.H. Cohen. *Ada as a Second Language (2nd edition).* McGraw-Hill, 1996.

[Cor04] Ada Core. *GNAT Reference Manual.* Ada Core Technologies, Inc., 2004.

[CP94] C. Comar and B. Porter. Ada 9x Tagged Types and their Implementation in GNAT. *Proceedings of Tri-Ada'94*, pages 71–81, 1994.

[Dew94] R. Dewar. The GNAT Compilation Model. *Proceedings of Tri-Ada'94*, pages 58–70, 1994.

[DIBM96] O. Dong-Ik, T.P. Baker, and S.J. Moon. The GNARL Implementation of POSIX/Ada Signal Services. *Reliable Software Technologies. AdaEurope'96*, LNCS 1088:275–286, June 1996.

[Dis92] G.J. Dismukes. Implementing Tagged Types and Type Extensions for Ada 9x. *TRI-Ada'92 Proceedings*, ACM, November 1992.

[GB92] E.W. Giering and T.P. Baker. Using POSIX Threads to Implement Ada Tasking: Description of Work in Progress. *TRI-Ada'92 Proceedings*, pages 518–529, ACM, November 1992.

[GB94] E.W. Giering and T.P. Baker. Ada 9X Asynchronous Transfer of Control: Applications and Implementation. *Proceedings of the SIGPLAN Workshop on Language, Compiler, and Tool support for Real-Time Systems*, 1994.

[GB95] E.W. Giering and T.P. Baker. Implementing Ada Protected Objects. Interface Issues and Optimization. *TRI-Ada'95 Proceedings*, pages 134–143, ACM, Anaheim, California, 1995.

[GMB93] E.W. Giering, F. Mueller, and T.P. Baker. Implementing Ada 9X features using POSIX Threads: Design Issues. *TRI-Ada'93 Proceedings*, pages 214–228, ACM, Seatle, Washinton, September 1993.

[GWEB83] G. Goos, W.A. Wulf, A. (Jr.) Evans, and K.J. Butlet. An Intermediate Language for Ada. *Lecture Notes in Computer Science*, (161), 1983.

[MAAG96] J. Miranda, A. Alvarez, S. Arevalo, and F. Guerra. Drago: An Ada Extension to Program Fault-Tolerant Distributed Applications. *Reliable Software Technologies. Ada-Europe'96*, pages 235–246, June 1996.

[MAGA00] J. Miranda, A. Alvarez, F. Guerra, and S. Arevalo. Programming Replicated Systems in Drago. *International Journal of Computer Systems: Science and Engineering*, 15(1):49–59, June 2000.

[MGMG99] J. Miranda, F. Guerra, J. Martin, and A. Gonzalez. How to Modify the GNAT Front-end to Experiment with Ada Extensions. *Reliable Software Technologies. Ada-Europe'99*, pages 226–237, June 1999.

[SB94] E. Schonberg and B. Banner. The GNAT Project: A GNU-Ada9X Compiler. *Studies in Computer and Communications Systems. Ada YearBook.*, pages 147–158, 1994.

[Sta04] R.M. Stallman. *GCC Manual*. Free Software Foundation, 2004.

[vK87] J. van Katwijk. *The Ada-Compiler: On the Design and Implementation of an Ada Compiler*. Technische Universiteit Delft, Amsterdam, September 1987.

Index