The GNAT Project: A GNU-Ada9X Compiler

The GNAT team*
gnat-report@cs.nyu.edu
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York, NY 10012

Cyrille Comar
comar@cs.nyu.edu
Sup'Aéro and
New York University
Computer Science Dept.

Franco Gasperoni
gasperon@inf.enst.fr
Télécom Paris – ENST
Département Informatique

Edmond Schonberg
schonber@cs.nyu.edu
New York University
Computer Science Dept.

Abstract

The GNAT project at New York University is building a high-quality Ada9X compiler, to be distributed free and with sources, following the successful mechanisms established by the Free Software Foundation for the GCC compiler. GNAT will allow students, academics, and software professionals to experiment as early as possible with the new version of Ada. GNAT will also help the spread of Ada to the software community at large.

*Robert Dewar and Edmond Schonberg are the principal investigators on the project. The team is based at New York University, and includes active participants from a number of other institutions (listed below): Bernard Banner, Cyrille Comar, Sam Figueroa, Richard Kenner, Brett Porter, Gail Schenker (all at NYU), Franco Gasperoni (Telecom Paris), Ted Giering (Florida State University), Paul Hilfinger (UC-Berkeley), Yvon Kermarrec (Telecom Bretagne), Laurent Pautet (Telecom Paris) and Jean-Pierre Rosen (Adalog).

Contents

1 Introduction: Ada9X and the GNAT Project

The Ada community has proposed a number of explanations for the relative lack of success of Ada vis-a-vis of C and more recently C++, in spite of the clear superiority of Ada as a language for software engineering. At least one reason for the slow spread of Ada through the software community has been the absence of a cheap (or even free) high-quality compiler that can run on a variety of platforms and is usable both for training and serious software construction. The issue of training is a particularly critical one: students (and universities) cannot afford expensive programming environments, and the choice of programming languages for teaching is often ruled by cost considerations. The widespread use of C is in part due to the ubiquitousness of UNIX. The recent successes of C++ are at least in part attributable to the availability of Turbo-C++ on PC's, and of course G++ (the GCC C++ compiler) on UNIX platforms.

The imminent introduction of Ada9X presents us with a new opportunity. The language [**?**] offers up-to-date tools for object-oriented programming, for information systems, for distributed systems, for interfacing with other languages, for hierarchical system decomposition, etc. If a free, high-quality compiler were to appear at the same time as the standardization of the language is completed, it would assist considerably in spreading the knowledge of the new language, and in encouraging comparisons with existing languages (in which we can expect Ada9X to show its superiority).

The GNAT project aims to produce such a compiler. GNAT (an acronym for GNU NYU Ada Translator), is a front-end and runtime system for Ada9X that uses the successful GCC back-end as a retargettable code generator. GNAT is thus part of the GNU software, and is distributed according to the guidelines of the Free Software Foundation. GNAT will be a complete compiler, but will not be validaded by New York University. In fact, GNAT will be available before validation procedures for Ada9X compilers are completed, because timeliness is crucial to its mission. Preliminary versions of GNAT, albeit very incomplete, are already being distributed, and are contributing to the diffusion of the language. The availability of sources for the system is allowing language designers and implementors to participate in the writing of GNAT itself. Compiler constructors are also benefiting from the existence of a reference implementation for new language constructs. We give below information on how to obtain GNAT and how to participate in the community effort of completing and improving it.

The next section describes the GCC compiler system. Next we summarize the structure of GNAT. Following sections discuss some details of the front-end and code generator. We then present what is probably the most innovative aspect of GNAT, namely the library mechanism. We then discuss the binder, and conclude with a status report on the completion

and performance of the system.

2 GCC: An Industrial-Strength Compiler

GCC 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 then redistributes the modified product to supply the sources for the modifications as well. In this fashion, enhancements to the original software benefit the software community at large [?].

GCC is today the centerpiece of the GNU software. GCC is a retargetable and rehostable 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++, Modula-3, Fortran, Objective-C, and most recently Ada. Technically, the crucial asset of the GCC is its mostly language-independent, target-independent code generator, which produces code of excellent quality both for CISC machines such as the Intel and Motorola families, as well as RISC machines such as the IBM RS/6000, the DEC Alpha, or the MIPS R4000. 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. Most of the code generation and optimization then uses the RTL, which GCC maps when needed into the target machine language. The leverage of constructing a front-end for GCC is thus enormous: GNAT potentially has over 30 targets, and runs already on more that half-a-dozen of them. An Ada9X cross-compiler for a Motorola real-time controller chip was built in a few days using standard GCC configuration tools for cross-compilation. Furthermore, GCC produces high-quality code, comparable to that of the best commercial compilers.

3 The Organization of GNAT

3.1 Introduction

The first decision to be made was the language in which GNAT 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. We 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, we were able to bootstrap the compiler, and abandon the commercial compiler we had been using up to that point. As Ada9X features are implemented, we are now able to write GNAT in Ada9X. In fact,

the definition of the language depends heavily on hierarchical libraries, and cannot be given except in Ada9X, so that it is natural for the compiler and the environment to use child units throughout.
Figure ?? shows the overall structure of the GNAT system. The front-end of the GNAT

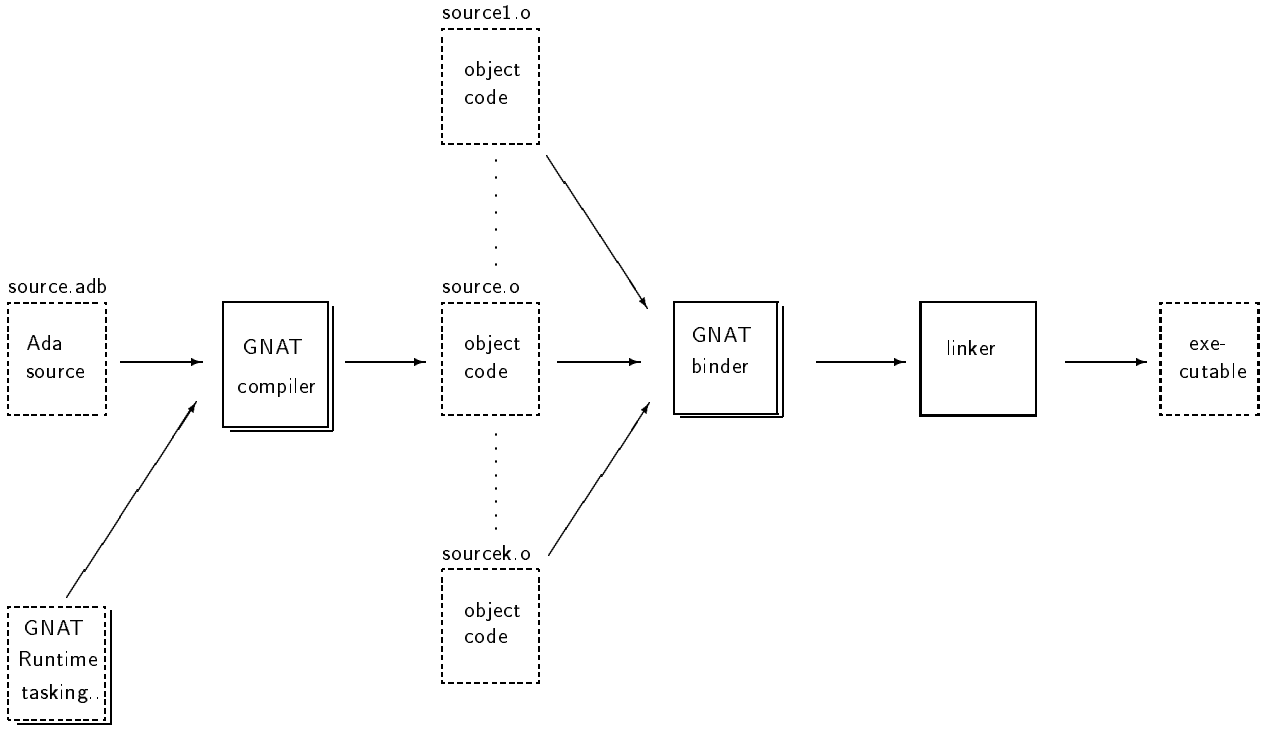

Figure 1: The structure of GNAT: compiler, binder, runtime.

compiler is thus written in Ada9X. The back-end of the compiler is the back-end of GCC proper, extended to meet the needs of Ada semantics.

The front-end comprises four phases, which communicate by means of a rather 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. The layering of the system, and the various levels of abstraction, are the obvious benefits of writing in Ada, in what one might call "proper" Ada style.
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 directy in the AST. The GNAT structures are thus close in spirit to those of DIANA [?], albeit more compact. It appears that the AST will be adequate to support an ASIS interface [?].

The four phases of the compiler are sketched in figure **??**.

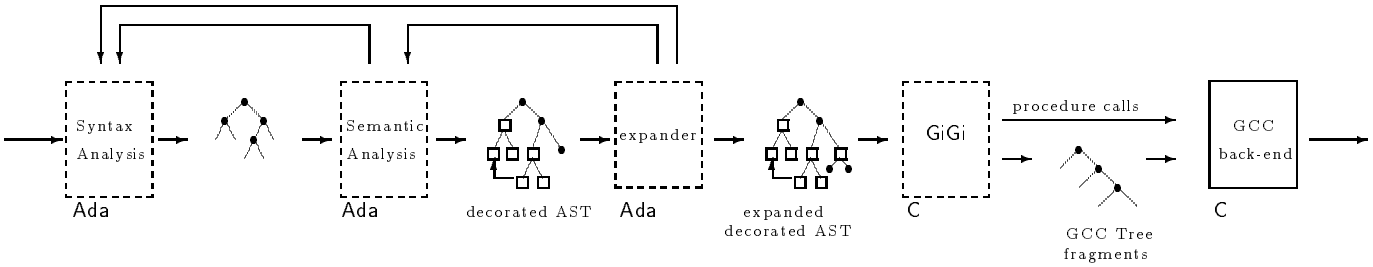

Figure 2: Phases of the GNAT compiler.

Subsequent sections describe each of the phases in detail.

GNAT includes three other modules which are not involved in code generation but are an integral part of any Ada compilation system.These are the runtime and tasking executive, the library manager, and the binder.

3.2 Syntax Analysis

The parser is a hand-coded recursive descent parser. It includes a sophisticated error recovery system, which among other things takes indentation into account when attempting to correct scope errors. In our experience, the recovery is superior to that of other compilers, and the parser is remarkably stable in the presence of badly mangled programs. All GNU compilers heretofore had used LALR(1) parsers generated with Bison (The GNU equivalent of YACC). The choice of a handwritten parser at this date may seem surprising, but is amply justified by the following:

Clarity. The parser follows carefully the grammar given in the Ada9X reference manual. ([**?**]). This has clear pedagogical advantages, but precludes the use of a table-driven parser, given that the grammar as given is not LALR(k).

Error messages. The most important reason is the quality of the error reporting. Even in case of serious structural errors, such as an interchange of "`;`" and "`is`" between specification and body of a subprogram, GNAT generates a precise and intelligible message. Bottom-up parsers have serious difficulties with such errors.

Performance. Even though the overall performance of the system is bounded by the speed of the code generator, it does not hurt that the parser of GNAT is faster than any table-driven one.

3.3 Semantic Analysis and Expansion

These two interlinked phases have the following purpose:

Semantic analysis performs name and type resolution, decorates the AST with various semantic attributes, and as by-product performs all static legality checks on the program.

The expander modifies the AST in order to simplify its translation into the GCC tree. Most of the expander activity results in the construction of additional AST fragments. 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. This recursive structure is carried further: some predefined operations such as exponentiation are defined by means of a generic procedure. The expansion of the operation results in the generic instantiation (and corresponding analysis) of this generic procedure.

There is a further unusual recursive aspect to the structure of GNAT. The program library (described in greater detail below) does not hold any intermediate representation of compiled units. As a result, package declarations are analyzed whenever they appear in a context clause. Furthermore, if a generic unit, or an inlined unit `G`, is defined in a package `P`, then the instantiation or inlining of `G` in the current compilation requires that the body of `P` be analyzed as well. Thus the library manager, the parser, and the semantic analyzer can be activated from within semantic analysis (note the backward arrows in figure **??**).

3.3.1 Type Resolution

Type and overload resolution is performed by means of the well-known two-pass algorithm. During the first (bottom-up) pass, each node in a complete context is labelled with its type, or if overloaded with the set of possible meanings of each overloaded reference. During 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. When resolving a call to a primitive operation of a tagged type, the second pass also determines the actual in the call that is to serve as controlling argument of the dispatching call.

3.3.2 Expansion Activities

The modifications performed by the expander are tree transformations that must be applied to those Ada constructs that do not have a close equivalent in C, such as allocators, aggregates, tagged types and dynamic dispatching, and all aspects of the tasking. The

expansion phase also simplifies some aspects of semantic analysis which are awkward to perform strictly in one pass, eg. the correct handling of the private part of a package declaration. The most important expansions are the following;

1. Construction of initialization procedures for record and array types, and invocation of these procedures for each object of such a type. This is also done for tasks and protected objects.

2. Generic instantiation. Instantiation is always done in-line, so that declaration and body of the instance are inserted into the AST at the point of instantiation.

3. All tasking operations are transformed into calls to subprograms in the run-time system. The recursive mechanisms of GNAT are particularly useful here. For example, consider operations on the attribute `COUNT`. The run-time holds the specification of a run-time function that examines the corresponding queue. Rather than including the details of such a function in the compiler proper, the run-time package is analyzed by the compiler as if it had appeared in the context clause of the current compilation. If the function is subject to an `INLINE` pragma, the compiler can perform the inlining as well, without forcing the compiler to have detailed information about the run-time, and without affecting code quality. Such flexibility cannot be achieved with a more conventional compiler organization. Because of the speed of the compiler, the cost of this approach in terms of space and time is comparable or cheaper than the conventional approach.

3.4 Gigi and Code Generation

The phase labeled Gigi (Gnat to Gnu) interfaces the front-end with the GCC code generator. Gigi traverses the decorated and expanded AST, in order to build the corresponding GCC tree, which is then input to the code generator proper. More precisely, the activities of GCC tree construction and code generation are interspersed, so that after each code generation activity, the GCC tree fragment can be discarded. At no time is a full tree built (there is no such notion in GCC). This is in line with the one-pass model of compilation used for C, and is memory-efficient.

In order to bridge the semantic gap between Ada and C, several code generation routines in GCC have been extended, and others added, so that the burden of translation is also assumed by Gigi and GCC whenever it would be awkward or inefficient to perform the expansion in the front-end. For example, there are code generation actions for exceptions, for variant parts, and for access 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.

3.4.1 Discriminated Records and Dynamic Arrays

Discriminated records are implemented without hidden pointers: if the position of a record component depends on a discriminant (for example if the size of a previous component depends of a discriminant) then GCC generates inline code to compute the address of the component, rather than storing offsets in the object.

the implementation of objects whose size is dynamic makes use of so called fat pointers. A fat pointer is a record with two components: a pointer to an object, and a pointer to a descriptor that contains bounds information on the object. Most accesses to such an object make use of the descriptor. GCC builds fat pointers when needed, for example when passing a composite type in a call to a formal parameter that is an unconstrained type.

3.4.2 Exceptions

The exception mechanism is intended to be usable by all GCC languages that have exceptions: Ada, C++, and Modula-3. The mechanism should be sufficiently uniform to allow multi-language programs to function in the presence of language-specific exceptions and exception handlers: for example, an Ada exception may propagate from a C++ module to an Ada handler. The mechanism should also be zero cost, that is to say, there should be no run-time cost attached to the mere presence of a handler, only to the actual occurrence of an exception.

The design of exception handling is closely related to the semantics of finalization. Recall that on exit from any construct that declares some entities, there may be cleanup actions to perform: finalization of controlled objects, reclamation of local heap-allocated objects, etc. We implement this sequence of actions by means of a single chain that holds all local objects that may require finalization, and a single procedure that traverses this chain and invokes the appropriate finalization for each object therein. When an exception is raised, the stack must be unwound, and the finalization routines attached to each frame must be invoked in turn. The exception manager must be able to locate the exception handler, and then repeatedly unstack a frame and invoke its finalization procedure. The exception manager uses two tables for this purpose: an unwind table and a handler table.

Unwind Table. The linker builds a table of address ranges, each of which is either under the control of a given exception handler, or has an attached finalization procedure. The table stores the addresses of each.

Handler table. This table holds the list of exceptions managed by a particular handler.

The method depends on being able to find, without additional structures, the subprogram that contains the instruction that raised an exception. To insure that the processing

is language-independent, the cleanup procedure is parameterless, and only its address needs to be retrievable.

Exception propagation proceeds as follows:

Locating the handler. During this phase the exception manager uses the PC of the exception-raising instruction to locate the innermost active subprogram that has an applicable handler. This traversal of the stack is done without unwinding actions, so that the debugger can be invoked on the offending instruction in its proper context, in case there is no applicable handler.

Cleanup. In the second phase, stack unwinding takes place, and the unwind table is used to retrieve the cleanup procedures at each step, until the exception handler takes control.

3.5 Object-Oriented Programming

One of the most eagerly awaited aspects of Ada9X is its support for object-oriented programming. In this section we review briefly the novel approach of Ada9X to this important programming paradigm, and some GNAT implementation details. We examine in succession the three critical notions: inheritance, polymorphism, and dynamic dispatching.

3.5.1 Inheritance

What other object-oriented languages call objects are defined in Ada9X by means of tagged types. A tagged type is a record with a special component, called the tag, which governs dispatching. Tagged types can be extended with additional components. The notion of type extension, as well as the concept of inheritance of operations, are generalisations of the Ada83 mechanism of type derivation. GNAT implements tagged types by following closely the implementation of regular records. The expander transforms tagged types into records according to the following schema:

```
type R1 (D1, D2 : Type_D) is tagged
  record
     C1, Cn : Type_C;
  end record;
```

```
type R1 (D1, D2 : Type_D) is
  record
     _tag : Ada.Tags.Tag;
     C1, Cn : Type_C;
  end record;
```

Extensions are transformed as follows:

```
type R2 (D3, D4 : Type_D) is
  new R1 (D3, X) with
    record
       E1, En : Type_E;
    end record;
```

```
type R2 (D3, D4 : Type_D) is
  record
     _parent : R1 (D3, X);
     E1, En : Type_E;
  end record;
```

The components _parent and _tag use "_" as prefix to avoid potential name conflicts with user-defined components. After this transformation, any reference to an inherited component is turned into a reference to the embedded _parent.

3.5.2 Polymorphism

Polymorphism denotes the capability of treating in similar fashion objects that belong to a class of types. Classwide types serve to denote objects that can belong dynamically to any derivation class.

R1'Class is a type that is implicitly defined when R1 is defined, and which covers R1 and all its extensions. All operations on R1 can be applied to a value of type R1'Class. The implementation of classwide types is delicate, because a value of such a type has an indefinite subtype, that is to say an unknown number of discriminants and unknown components beyond those inherited from R1. We have found it convenient to define a classwide type as an extension with unknown typeless storage. For instance, when the expression new R1'Class (V) is encountered, GNAT will the foloowing type

```
type R1__Class_Subtype is record
   _parent : R1 (V.D1, V.D2);
   _extension : Array_Of_Bits (1 .. V'Size -
R1'Size);
end record;
```

There is another delicate point concerning the implementation of classwide types. All members of the class must have a compatible layout, so that offsets of corresponding components must be identical. This conflicts with the need to place discriminants at fixed offsets, usually at the beginning of the record, so as to be able to calculate the placement of components that depend on those discriminants. If any descendant can add new discriminants to a tagged type, it is not possible to make discriminants contiguous. Figure ?? shows the layout for the types of the previous example: we are forced to place D3 and D4 between _parent and the components E1, En.

3.5.3 Dynamic Dispatching

Primitive operations that are inherited by a type extension can be redefined, in which case the new definition overrides the old one. When a primitive operation of the root type is applied to a classwide argument, the tag of the argument determines the implementation of the operation which is to be executed, i.e. the original operation or one of its overridings. The tag is implemented as a pointer to a dispatch table. The table contains pointers to the primitive operations of the type. There is one table for each tagged type, and the layout of all types in a derivation class is compatible, in the sense that different overridings of the same operation appear in the same table position. Note

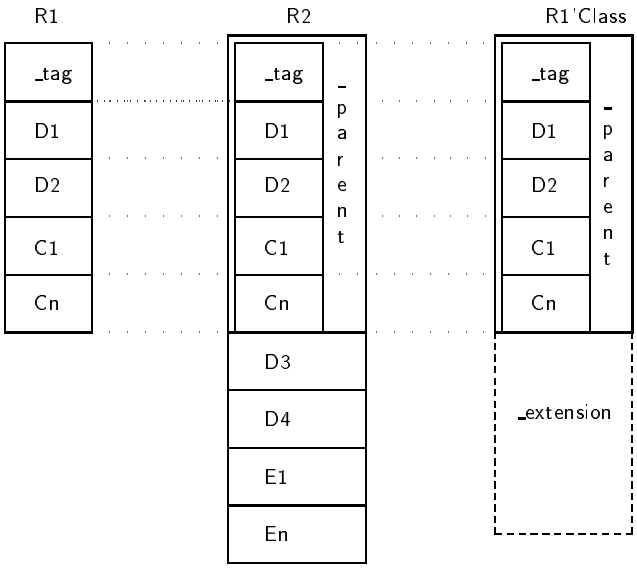

Figure 3: Storage for classwide types

that in Ada terms this table is not an array, because each component is an access to a subprogram with a different profile. A call to a primitive operation is dispatching if the specific type of its tagged arguments cannot be determined statically. In such a case, the tag of one of the actuals is chosen to determine which subprogram to call. Simplifying somewhat,if we consider type `R1` introduced in section **??** and we encounter

```
Primit_n (Param:  R1'Class)
```

then dispatching amounts to replacing:

```
Primit_n (X)    with    X._tag.all.Access_Primit_n.all (X)
```

The layout of the dispatch table is shown in Figure **??**. The first two components of the table simplify the implementation of the membership operation for tagged types.

3.6 The Runtime: GNARL

The most important activities of the run-time have to do with task management: creation, activation, rendez-vous, termination. The runtime maintains the data structures needed to manage, schedule, and synchronize tasking activities. In order to make GNAT easily portable, the runtime is written in Ada (with some very small assembly glue) and two procedural interfaces, GNARLI and GNULLI, are used to isolate the compiler from the runtime, and the runtime from the underlying operating system.

GNARLI (GNAT run-time library interface) is the interface between the compiler and the run-time. Each Ada construct that applies to tasks or protected objects is implemented by one or more subprograms in the run-time. The expander transforms each occurrence

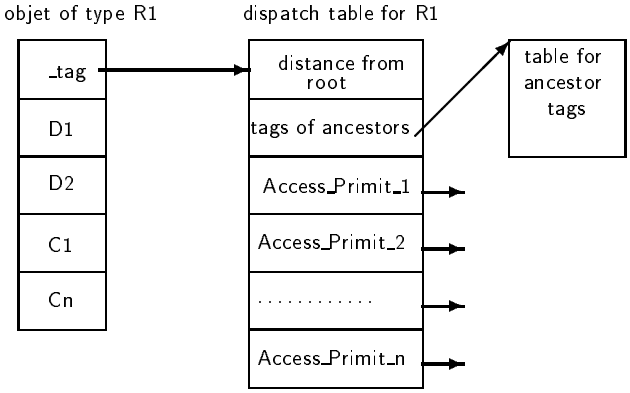

Figure 4:

of such constructs into the corresponding series of calls. The packages that constitute the run-time are treated as any other unit of the context of the compilation, and analyzed when needed. This obviates the need to place run-time information in the compiler itself, and allows a knowledgeable user to modify the run-time if he/she so chooses. The design of GNARL is based on the CARTS (Common Ada Run-Time System) specification [**?**].

GNULLI (GNAT low-level library interface) provides the interface between the run-time and the underlying operating system. The design of GNULLI makes use of a few POSIX threads primitives, and assumes the existence of such primitives in the host OS. A threads package that emulates those primitives is supplied for systems that do not have them, e.g. conventional Unix systems. Otherwise the implementation of GNULLI is straightforward on modern operating systems such as Solaris, Mach and OS/2.
The design and implementation of GNARL have been carried out at Florida State University by the group directed by Ted Baker and Ted Giering, and follows their design of previous protable Ada runtimes, notably CARTS and MRTSI.

3.7 Library Management

The notion of program library is seen justifiably as 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 all Ada compilers to date, the library is a complex structure that holds intermediate representations of compiled units, information about dependences between compiled units, symbol tables, etc. The ARM strongly suggests that such a structure is mandatory, but in fact a monolithic library is not required to implement rigorously the semantics of separate compilation. Furthermore, the monolithic library approach is ill-adapted to multi-language systems, and has been responsible for some of the awkwardess of interfacing Ada to other

languages.

We have chosen a completely different approach in GNAT. 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 the declarations of the units appearing in the context clause of a given compilation are always analyzed anew. Dependency information is kept directly in the object files, and amounts to a few hundred bytes per unit. The binder can be used to verify the consistency of a system before linking, and is also used to determine the order of elaboration. Given the speed of the front-end, our approach is no less efficient than the conventional library mechanism, and has three important advantages over it:

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

2. Given that object files only depend on sources, not on other objects, there is no longer a required order of compilation. All the components of a system can be compiled in any order. Only the modification of a source program may obsolete a compiled unit. A well-known dreaded phenomenon of previous Ada systems, namely the accidental recompilation of one unit that obsoletes a slew of other units in the library, even when the source is unchanged, is thus avoided completely.

3. Inlining works in a much more flexible way than in normal compilers. Given that compiling, and thus 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.

It is gratifying that this flexible model is fully conformant with the prescribed semantics given in the ARM, and at the same time confortable for programmers used to the behavior of `make` and similar tools. The GNAT model simplifies the construction of multi-language programs and makes Ada look more familiar to programmers in other languages.

3.8 The Binder

The role of the binder is twofold:

- It verifies the consistency of the objects that are to be assembled into an executable.
- It determines a valid order of elaboration for the units in the program, and packages the calls to the corresponding elaboration procedures into a single subprogram, to be invoked before the main program.

The binder makes use of the information created when each unit is compiled. This information includes the semantic dependencies of each unit, the date of latest modification of their sources, the presence of various elaboration pragmas, and whether a given unit may be a main program.

The binder has been designed with flexibility in mind. In one mode, it can verify that all objects depend on a consistent set of sources. Given that time stamps of the sources used for a compilation are kept in the object files, this check does not require that sources themselves be present, which is an advantage in commercial settings for software distribution.

Another mode of operation is to verify that the system is up-to-date, that is to say that no source was modified after compilation. In all cases, possible inconsistencies are diagnosed and treated as fatal errors. There are however cases in which this is undesirable. For example, it is irritating to be forced to recompile a large system only because comments were added to a low-level package on which many units depend. An additional option instructs the binder to ignore time stamps and create an elaboration procedure unconditionally. Such an option requires precise understanding on the part of the user, and is certainly not safe, but it may be indispensable in certain circumstances, and will be welcome by experienced programmers.

Finally, the binder is intended to work with other GCC languages, and can produce different output programs. By default, the object file given as input is taken to be the main program. In this case, the binder builds a C file containing the function `main`, which is the mandatory main program for a C compilation. This function consists of a series of calls to elaboration procedures, followed by a call to the main Ada program. The intended main program may not even be in Ada, in which case the binder output consists solely of the elaboration calls.

4 Conclusion

Compiler quality means different things to different users. For students and beginners, GNAT intends to be user-friendly, provide lucid error messages, and fast turn-around for small programs. For a software engineer, code quality is paramount, and GNAT can rely on the proven performance of the GCC back-end. For the embedded-systems developer, the existence of cross-compilation tools is critical, and here as well GCC provides the necessary functionality. For the language researcher and the compiler writer, the existence of sources of a full compiler is invaluable.

GNAT has no size limitation, beyond that imposed by the full memory of the host machine. The speed of the system is substantial: on a 66-Mhz i486 machine, the front-end runs at 40,000 lines/min., and the full compiler at 8,000 lines/min. Such a performance is comparable to that of the best commercial compilers, and is likely to improve by a factor

of two when various tracing options are removed and full inlining is supported.

To date (Dec. 1993) GNAT is sufficiently complete and robust to compile itself (around 110,000 lines of Ada), compile GNARL, and pass hundreds of ACVC tests. Nevertheless, given the size and complexity of Ada9X, we know that a few person/years are still required to complete the task. In spite of its incompleteness, the GNAT system already has a small but dedicated set of users. The cooperative spirit fostered by the activities of the Free Software Foundation is striking: days after the first release of GNAT, several ports to unexpected machines were reported, and offers were made to the project of important software components: bindings to Mach, to X-windows, implementation of the information systems annex, etc. This synergy within the Ada community is a rewarding byproduct of the GNAT project.

5 Appendix: How to Obtain GNAT

GNAT is available by anonymous ftp from cs.nyu.edu, directory pub/gnat. This directory contains a README file, sources files, and binaries for OS/2 and for Sparc/SunOS. Installation on other targets currently requires that GCC already be installed.
A mailing list is maintained at New York University to notify users of new version releases. Send mail to gnat-reque@cs.nyu.edu to be included in this list.
Users are invited to submit bug reports to gnat-report@cs.nyu.edu. Information on new ports, on enhancements to the compiler, as well as other software contributions, are welcomed at the same electronic address.

Acknowledgements. The work we have described is the result of the collective efforts of the GNAT team, and we thank all of them for the pleasure of working together. We thank Richard Stallman, not only for the GCC system, but for his seminal insight that the Ada library model could be source-based. We also want to thank Ted Baker (Florida State University), Bruno Leclerc (Télécom Bretagne), and Tucker Taft (Intermetrics) for innumerable advice and discussions.

References

[1] J.B.Bladen et al., *Ada Semantic Interface Specification (ASIS)*, Conference Proceedings, TriAda'91, San Jose,California, October 1991.

[2] Ada9X Mapping/Revision Team, *Programming Language Ada–Language and Standard Libraries*, Draft, Version 4.0, Intermetrics, September 1993.

[3] T.P.Baker, and E.W.Giering, III, *Implementing Ada9X features using POSIX threads: design issues*, Conference Proceedings, TriAda'93, Seattle, Washington, September 1993.

[4] G.Goos, W.A.Wulf, A.Evans, Jr., and K.J.Butler, *DIANA - An Intermediate Language for Ada*, Lecture Notes in Computer Science, number 161, Springer-Verlag, 1983.

[5] R.M.Stallman, *Using and Porting GNU CC*, Free Software Foundation, December 1992.