The GNAT Project: A GNU-Ada 9X Compiler

Edmond Schonberg

New York University
Courant Institute
251 Mercer Street
New York, NY 10012
schonberg@cs.nyu.edu

Bernard Banner

New York University
Courant Institute
251 Mercer Street
New York, NY 10012
banner@cs.nyu.edu

Abstract

We describe the general organization of the GNAT compiler, its relationship to the GCC multilanguage compiler system, and some of the architectural details of the system. This paper serves as an introduction to the following papers in this session.

Introduction: Ada 9X 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 imminent introduction of Ada 9X 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 Ada 9X 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 Ada 9X that uses the successful GCC [GCC94a] back-end as a retargetable 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 validated by New York University. Successively more complete versions of GNAT have been distributed over the last 18 months. The availability of sources for the system is allowing language designers and implementors to participate in the construction of GNAT itself. As a result, many members of the Ada community are contributing software packages for portions of GNAT, in particular for the language annexes.

In the following sections we give a technical overview of the structure of GNAT. We then discuss some of the more interesting features of the system, from the user point of view. Finally, we describe the status of the annexes, the GNU licenses and some of the ongoing collaborations involving the GNAT team and others.

1. 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 General 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.

2. The organization of GNAT

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 Ada 83, and in typical fash-

This work has been supported in part by the Ada 9X Program Office through Air Force Contract FO8630-92-C-0019.

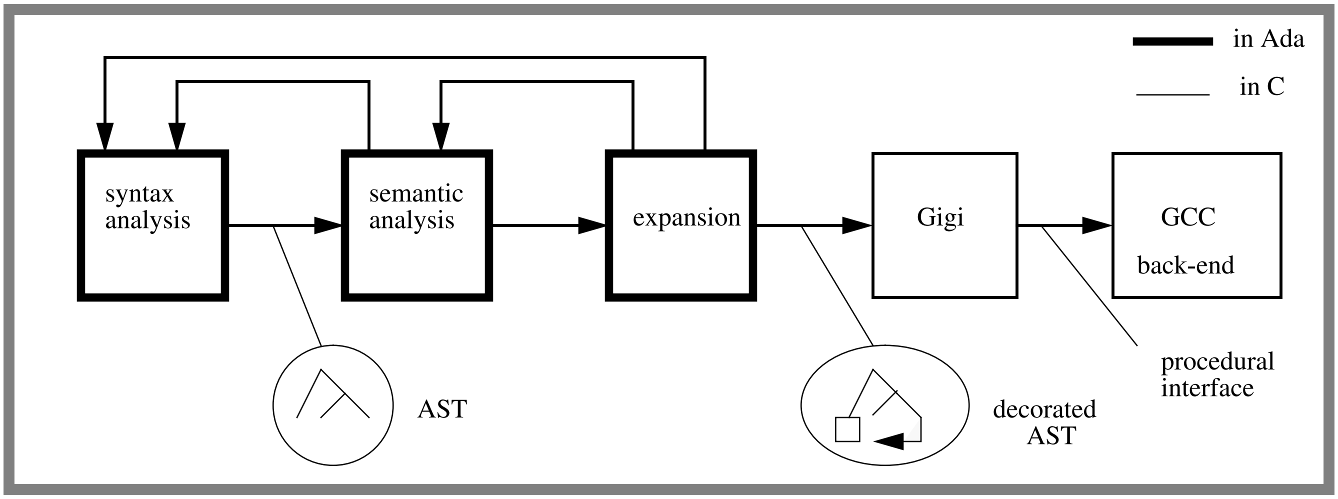

Figure 1

ion 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 Ada 9X features are implemented, we are now able to write GNAT in Ada 9X. In fact, the definition of the language depends heavily on hierarchical libraries, and cannot be given except in Ada 9X, so that it is natural for the compiler and the environment to use child units throughout. Figure 1 shows the overall structure of the GNAT compiler.

The front-end of the GNAT compiler is thus written in Ada 9X. 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 directly in the AST. The GNAT structures are thus close in spirit to those of DIANA [DIANA83], albeit more compact. It appears that the AST will be adequate to support an ASIS interface, [ASIS91] but we have not examined the design of such an interface in any detail yet.

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. These components are also written in Ada 9X.

Before presenting some details about the compiler proper, we want to highlight two user-friendly aspects of the GNAT system that are particularly relevant to its use in the early days of Ada 9X.

2.1 A compiler for everyone: character sets

One interesting aspect of GNAT is its support for foreign language character sets. GNAT fully implements the new standard in this respect, and in fact goes well beyond the stated minimum requirements.

Latin-1 is fully supported, including its use in identifiers, as described in the RM. In addition, optional switches allow localization of the identifier character set to Latin-2, Latin-3, Latin-4, the IBM PC set, the restricted (Ada 83) 7-bit set, or the full upper half without case equivalence (useful for some languages like Thai, where there is no upper/lower case equivalence). Types wide character and wide string are fully supported, and a compiler switch allows the use of wide characters in identifiers. Several different encoding methods are implemented for wide characters, including the EUC and Shift-JIS encodings in common use in Japan. We envision adding in the future other representations such as an interface to the Unicode encoding scheme.

2.2 A compiler to ease the transition: 83/9X modes

Two modes of compilation are offered in GNAT. Ada 83 mode allows programs to be compiled using Ada 83 rules. This is most useful for programs which might be incompatible with Ada 9X rules such as those using the new reserved words like **tagged** as identifiers in their programs. This mode will also flag as errors those constructs which are allowed in Ada 9X but not in Ada 83 such as basic declarative items appearing after later declarative items, unconstrained variable declarations, among others. Special recognition of Ada 9X constructs takes place so that when new features (tagged types or requeue statements, say) are encountered in Ada 83 mode, appropriate error messages are issued, instead of treating them as unrecognizable text (as would be the case for a typical Ada 83 compiler).

Of course the Ada 9X mode is the default. While we have tried to be faithful in our Ada 83 mode, the introduction in Ada 9X of some real upward incompatibilities makes it very hard to remove all 9X semantics from the implementation, so that programs that GNAT considers legal Ada 83 may in fact include some minor 9X extensions. Examples of these include the use of the Ada 9X Text_IO, which has added Append_Mode to File_Type. Given that our chief goal is to provide an *Ada 9X* compiler, these small departures from strict Ada 83 behavior are benign. We envision having a mode which will warn users of constructs in their Ada 83 programs which might introduce incompatibilities in Ada 9X along the lines discussed in [PLOEDEREDER92].

3. Syntax analysis in GNAT

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. This is especially important in a student oriented setting. 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:

(a) **Clarity**. The parser follows carefully the grammar given in the Ada 9X 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).

(b) **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.

(c) **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.

3.1 Some friendly error recovery

It is expected that many of the people that will evaluate GNAT will be users of other languages like C and Pascal. The error detection of GNAT recognizes some of the classical mistakes that these users are expected to make, such as using **!=** or **<>** for inequality and **==** for equality; it informs them that the proper token should be **/=** or **=**.

The parser tries to make use of the casing of identifiers in the program, and assumes that the user has some consistent policy regarding the casing convention used to distinguish keywords from user-defined entities. The parser deduces this convention from the first keyword and identifier that it encounters. In the example given in figure 2, the compiler makes use of the upper/lower case rule for identifiers to treat **exception** as an intended identifier rather than the beginning of an exception handler.

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

Figure 2

Special treatment is given to particularly common mistakes such as using a positional aggregates with a single component (instead of the required named association), as shown in figure 3. Needless to say, this recovery requires semantic information:

```
1. procedure Wrong2 is
2.    type R is record
3        M : Integer;
4.    end record;
5.    Vr : R;
6. begin
7.    Vr := 1;
            |
  expected type "r" declared at line 2:9
  found type "universal_integer"

8.    Vr := (1);
             |
    positional aggregate cannot have one component.
9. end;
```

Figure 3

The next example given in figure 4 makes use of indentation to match if statements.

```
1. procedure Wrong3 is
2. begin
3.   if a > b then
4.      null;
5.
6.end;
   |
   "end if;" expected for "if" at line 3:4
```

Figure 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.

4. 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 a by-product performs all static legality checks on the program.

The semantic analysis follows the rule that all modifications to the AST allow recovery of the original source program. The expander is not bound by such a rule. As a consequence, the output of the semantic pass is the proper interface to an ASIS tool.

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, for which the run-time library holds instantiations for the various predefined types. The expansion of the operation results in the loading and compilation of the required instantiation, followed by a call to the corresponding function. Aggregates, initialization procedures, equality on composite types, boolean operations on arrays, and tasking operations are typical expansions.

4.1 Expansion: an example

A typical task of the expander is the construction of initialization procedures for composite types. For arrays, the initialization procedures expand into loops whose body invokes the initialization procedure for the component type. For records the initialization procedure invokes the corresponding procedures for each component type, or else assigns the default expression present in the record type declaration. In either case it is simpler to build the AST for the procedure, and invoke (recursively) the semantic analysis on this AST, which may lead to further expansion and analysis, rather than to laboriously generate a low-level representation for the operations involved.

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.

4.2 Invoking Run-Time routines

The possibility of analyzing other units in the middle of semantic analysis gives GNAT a powerful mechanism to interface with run-time routines without having to know their internals. The front-end has a map that associates various run-time library items with the library files that contain them. If the compiler needs to generate a call to one of these items (say, a tasking support routine) or make a reference to library data, it can retrieve the corresponding file and analyze it, thereby generating the proper external symbol in the user program. It is even possible for the compiler to inline run-time procedures in the user program without having to embed the details of the runtime in the compiler proper. For a compiler that is intended to be portable, and therefore depend on different run-time libraries, this is an obvious asset.

Two aspects of the semantic processing deserve additional discussion: the handling of generic units, and the implementation of tagged types and object-oriented features. The latter is described in [TAG94]. We proceed to describe briefly the implementation of generic units.

4.3 Generic units

Traditionally, there are two implementation techniques for generic instantiations: 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. Following the source-based model of the compiler, [LIB94], generic specifications are analyzed each time they appear in a context clause, and there is no code nor intermediate representation produced by such analysis. The analysis of the unit performs name resolution and all the required semantic checks, and the syntax tree is annotated with semantic information. There is no tree expansion performed on a generic unit, except insofar as some expansion is intrinsic to semantic processing (e.g. the transformation of a range attribute into a discrete range). The semantic analysis is performed on a copy of the original tree, which retains the unanalyzed structure of the parsed source text. After semantic analysis, all references in the generic unit that are not local are copied onto the original generic tree.

At the point of generic instantiation, the tree for the generic unit is copied, and the new tree is treated as a regular (non-generic) unit which is analyzed and for which code is generated. Note that name resolution has captured all non-local references at the point of generic declaration. These references are not analyzed anew. On the other hand, local references in the generic unit are resolved as local references in the instance. The mechanism can be pictured as below in figure 5.

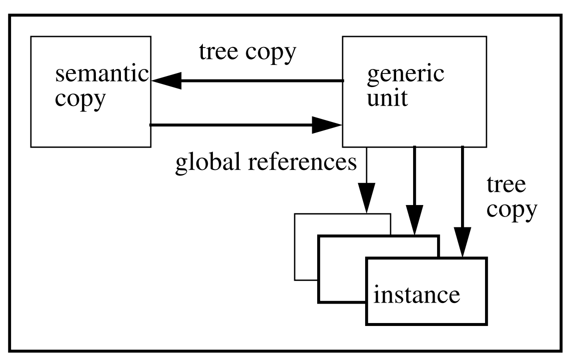

Figure 5

4.4 Generic bodies

The instantiation of generic bodies is done on a separate pass, after all semantic analysis of the main unit is completed. By delaying the instantiation of bodies, it becomes possible to compile legal programs that present difficult order-of-elaboration problems to other compilers. For example, consider the program fragments given in figure 6.

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. If the instantiation precedes textually the appearance of the generic body, then the instantiated body is placed immediately after the generic. Otherwise the instantiated body is placed in the tree at the point of instantiation.

5. 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. Such examples of this include discriminated records, unconstrained arrays, checked arithmetic and exceptions.A full description of the implementation of these is given in [GCC94b].

5.1 Two languages, one data structure

Gigi traverses the AST produced by the front-end, and emits calls to the generators that build GCC trees. The AST is constructed by an Ada program, and described by means of a series of Ada access functions. Given that Gigi is written in C, it must be provided with a view of the AST which is isomorphic with that held by the front-end. This is achieved by means of header files that are automatically generated from the Ada specifications for the AST. Several utility programs, which are part of the compiler-construction machinery, perform this task. These utilities are invoked whenever one of the AST specifications is updated.

5.2 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 and is explained in full detail in [GCC94b].

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 (GNU Ada 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

`package A is`
`  generic ...`
`  package G_A  is ...`
`end A;`
	`package B is`
	`  generic ...`
	`  package G_B is ...`
	`end B;`

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

Figure 6

expander transforms each occurrence 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.

GNULLI (GNU Low Level Interface) provides the interface between the run-time and the underlying operating system. The design of GNULLI makes use of a few POSIX-like 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, OS/2 and Windows NT. 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 portable Ada runtimes, notably CARTS and MRT-SI. The design of GNARL is described in detail in [RTS94]

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 Ada 83 RM strongly suggested that such a structure was 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 awkwardness 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 (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. 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:

(a) 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. 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.

(b) 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.

(c) The absense of any forced order of compilation facilitates the parallel compilation of a large system on different processors.

It is gratifying that this flexible model is fully conformant with the prescribed semantics given in the Ada Reference Manual, and at the same time comfortable 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. It even makes it possible to write multi-language programs with Ada components, whose main program is not itself written in Ada. Note that in such cases the binder will still be required, in order to construct the series of calls to the elaboration procedures of the Ada components.

The source-based organization of the library is the most novel aspect of the GNAT architecture. Details of the library mechanism and its use are found in [LIB94]

8. The Binder

The role of the binder is twofold: to check the consistency of the objects that are to be assembled into an executable, and to construct the program that is executed by the environment task. The binder determines a valid order of elaboration for the units in the program, and packages the calls to the corresponding elaboration procedures into a single procedure. This binding procedure then invokes the main program itself. The binding procedure includes the outermost exception handler that notifies the user when an otherwise unhandled exception terminates program execution.

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 and categorization 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 either as fatal errors or as warnings.

9. Evolving tools

Even though the goal of the project does not encompass a full programming environment, there are number of basic programming tools built around the compiler proper that exist or are under development.

9.1 Cross-Reference

The GNAT system provides a stand-alone tool, GNATF, which performs syntax and semantics checking only. This is considerably faster than using the full compiler since tree expansion and code generation are not performed. GNATF also generates cross reference and feature summary information.

The cross reference information produced falls into the following categories:

- Precise information about all declared entities, i.e. where they are defined and where they are used (traditional cross reference table).
- Warnings about entities declared but never used.
- Warnings about **with** clauses that are unnecessary, misplaced, or redundant.

Cross reference information can be gathered on entire programs, or on isolated units. Distinction is made between entities that appear directly in the compiled unit and those that are referenced in the compiled unit but are defined elsewhere.

The features summary facility of GNATF lists the names of Ada 9X features that are used in the source program, such as features restricted in Ada 83 (use of Latin_1, Ada 9X later declaration ordering) as well as major 9X constructs like child units, type extension and new pragmas. This will be useful in environments that want to control the introduction of Ada 9X features, and also to estimate the coverage of test suites such as the new ACVC suite.

9.2 Interfacing with the debugger

One of the critical features of GCC is the integration of all its translators with the symbolic debugger gdb. In order to make gdb usable with Ada, the front-end must generate the proper symbol table information, and gdb must become cognizant of various aspects of ada syntax and semantics. Work is under way to enhance gdb along the following lines:

- Understand Ada 9X data types, including arrays and discriminated records.
- Accept near-Ada syntax in expressions.
- Support setting of breakpoints and stepping-through in specific instantiations of generic units
- Permit "relatively convenient" reference to overloaded functions.
- Handle dispatching calls.

There should also be access to exception-handling and tasking, but this will be through explicit, ordinary gdb calls to designated runtime routines, and will not (initially) involve any enhancements to gdb itself.

9.3 A style checker

The GNAT front-end includes facilities to verify that the program obeys specific layout rules. For now, these are rules established for the text of GNAT itself, but we envision a system in which the user can specify various layout attributes that he/she wants enforced, such as:

- Indentation levels.
- Spacing requirements around tokens
- Uniform casing rules between definition and uses of identifiers
- Capitalization conventions for user-defined entities, reserved words and pragmas.
- Maximum line length.
- Presence of program unit names following the **end** keyword.

9.4 An Ada-specific make utility

The source-based organization of the compiler, and the dependency information used by the binder, can serve as the foundation for an intelligent Make program, which minimizes the amount of recompilation required after the modification of some component of a system. The design of such a tool is described in [FAST94].

10. Some statistics

The system is currently composed of 200,000 lines of Ada text and comments, distributed among 642 files. The parser is 18,000 lines, semantics comprises 37,000 lines, the expander 16,000, child units of System (including the tasking library) take 25,000 lines. The scanner, utilities, predefined libraries (child units of Ada) complete the count. Gigi comprises 10,000 lines of C. Roughly half of the Ada source lines are internal documentation and blank lines. The GCC back-end component of GNAT is approximately 220,000 lines of C. GNAT compiles its own front-end in about 20 minutes, on a 60 Mhz Pentium machine.

11. Conclusions

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.

Starting from its earliest, very incomplete releases, the GNAT system has had a growing and dedicated group 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 various language annexes, etc. This synergy within the Ada community is a rewarding by-product of the GNAT project.

Appendix I: How to obtain GNAT

GNAT is available via anonymous ftp from cs.nyu.edu (128.122.140.24), in the directory pub/gnat. This directory contains a README file, as well as the source for GNAT and binaries for various machines. These files can also be found on the Public Ada Library (PAL) and its mirror sites. Users are invited to submit any comments, bug reports or other communication to gnat-report@cs.nyu.edu.Those without Internet access can find GNAT on the various Ada CDROMs now being commercially distributed.

Appendix II: Status of Standard Libraries and Annexes

GNAT will eventually include a full implementation of the standard libraries described in Annex A, as well as all the specialized needs annexes. This section gives some details of the current status as of 8/94, and describes our technical approach.

Annex A: Predefined Language Environment

Most of the facilities in this section are already provided in GNAT in the form of standard Ada packages and subprograms. Some of the coding for the string functions was originally inspired by a version of the ADAR (Ada Decimal-Associated Reusabilia) package [ADAR94], but the code has been substantially revised to improve efficiency. Throughout, the intention is to achieve the best possible implementation, consistent with the requirement of maintaining maximum target independence. At the time of writing, the major omissions are the Input-Output packages other than Text_IO.

Annex B: Interface to Other Languages

The interface to C is fully implemented. We intend to complete the Fortran interface, including support for arrays stored in the Fortran column-major order. The interface to COBOL is more problematical, since many of the GCC targets lack significant COBOL compilers, and those that are available often have incompatible formats. As described in a subsequent section, we do intend to fully implement the information systems annex, with the intention of making Ada a viable language for fiscal calculations and other information systems tasks, in environments which often lack good compiler tools in this area.

Annex C: Systems Programming

This is substantially complete now, and all features will eventually be incorporated into GNAT. In particular, we will couple the package MACHINE_CODE capability to the existing support for code statements in GCC, and provide full support for all representation clauses and pragmas.

Annex D: Real-Time Systems

The complete implementation of the GNARL interface includes full support of all features in the real-time systems annex. This support is predicated on complete support of the GNULLI layer. On some systems, this may simply not be practical, and on those, we will not be able to meet some of the performance and behavior criteria of the Annex. For example, on a system like OS/2, which does not have asynchronous inter-task cancellation, we probably cannot provide ATC in a form that meets the requirements of this annex.

Annex E: Distributed Systems

There are three main facets of the implementation of the Distributed Systems annex: stream I/O, stub expansion for remote call interface packages, and the communication subsystem. The first is part of the core language, and its implementation is underway. The design of the stub expansion has been done by collaborators at Texas A&M and at ENST-Bretagne, and we plan to implement their design in GNAT. The same groups are also collaborating on the implementation of the communication subsystem, for which prototypes using TCP/IP and PVM already exist. We will provide an implementation of this annex on at least two targets: a network of SPARC workstations, and the IBM SP1 multiprocessor.

Annex F: Information Systems

We intend to provide a full implementation of the information systems Annex, including support of both binary decimal and packed decimal formats. The implementation of the picture editing packages will use standard Ada packages, with the possibility of specializing on machines providing special support for decimal operations or editing operations. We have chosen the Intel format for packed decimal, but this is defined in a single runtime module that is easily modified if some other format is preferred.

Annex G: Numerics

Most of the numerics capability is already supported in GNAT. We anticipate that accurate target-dependent support of the accuracy requirements will require significant modifications to the GCC backend, which at the moment does not do much worrying about floating-point accuracy (for example, the accuracy of conversion of floating-point constants is left to the assembler being used).

Annex H: Safety and Security

GNAT will include full support for the safety and security annex at least on certain specified machines. The hard part of the requirement is the need to provide fully annotated assembly listings. It is not very much in the Unix style for compilers to provide listings at all, let alone elaborately annotated listings, so it is not surprising that GCC has no capabilities in this area. One possible approach to the implementation is to write an external tool which uses the GDB interface to obtain debugging information (including line numbers, variable names etc.), and a modified version of GAS (the GNU assembler) to obtain the assembly listing with code and offsets

Appendix III: The GNU Licenses

Much confusion has arisen over the exact effect and restrictions of the GNU public license (common known as "copyleft"). This section is intended to clarify the import of copyleft. The GNAT compiler itself is covered by the GPL, the GNU General Public License [GPL91]. The intention of this license is to ensure that the compiler is freely available and freely copyable. It can still be sold, but the buyer can always copy it freely, so the general expectation is that it is sold for reasonable charges, like typical public domain (PD) software. Indeed one might ask why not simply place GNAT in the public domain to achieve this effect.

The trouble with PD software is that there is nothing to stop someone making (possibly minor) modifications, and then copyrighting the result and protecting it as fully proprietary software. This has happened to a number of important software products in the past. A government grant pays for the development of the software, which is initially in the public domain, and the PD version is still freely available. However, a commercial company forms, and continued maintenance applies only to a proprietary version, and soon everyone is paying for something that was intended to be free.

The GPL is precisely designed to preclude such an occurrence. It enjoins anyone that modifies GNAT to either keep the modifications entirely to themselves, or to distribute these modifications retaining the GPL, thus ensuring that the technology remains in the domain of free software. It is of course possible for companies to provide software support services, but no one can ever claim proprietary rights to GNAT or any program derived from GNAT.

Given the aim of producing a freely available Ada compiler, the use of the GPL makes good sense, and our contract with the government requires us to release the compiler under the GPL for this reason. However, there is often concern that the GPL will restrict what people can use the compiler *for* as opposed to what they can *do* with the compiler itself.

It is definitely *not* the case that any software that is generated by use of the GNAT compiler is "contaminated" by the GPL, and itself becomes freely distributable. Code generated by the compiler certainly is owned by whoever owns the source code. Library routines present a slightly different issue. If the library routines and runtime were covered by the GPL, then it would indeed be the case that all programs generated by the GNAT compiler would be covered by the GPL, and thus the use of the compiler would be greatly restricted. To avoid this, and to ensure the widest possible sphere of use for GNAT, the library and runtime are covered by the LGPL, the GNU Library General Public License [LGPL91]. This provides for free distribution of the runtime in programs, even if they are proprietary. The only requirement is that such programs be made available in a form that permits linking with different versions of the library routines, e.g. by distributing object files.

An example may prove more effective in making this point than a discussion of the legal details. NEXT Corp. uses GCC as the compiler technology for the NextStep system, using the Objective C front end. This front end, being married to the GCC backend, is covered by the GPL, and is freely distributed with sources. NextStep itself is certainly *not* freely distributed, and the sources are proprietary. (In fact, the NextStep sources and resulting proprietary executable operating systems are the primary capital of Next Inc.)

Although it is possible, as we have described, to use GNAT to create proprietary software, we remain committed to the ideal of free software, and hope that the full availability of GNAT with sources will encourage research and teaching using Ada, and will also encourage members of the worldwide Ada community to continue enhancing GNAT and thus to add to its value as a robust industrial tool in software development.

Acknowledgments

Because of space limitations we cannot mention the names of all of those whose messages, bug reports, suggestions for enhancements, and software contributions have added to the completeness and robustness of GNAT, but we must identify some collaborative efforts underway to complete the implementation of the various language annexes. First and foremost, a group at Florida State University (Ted Baker, Ted, Giering, Frank Mueller, Dong-ik Oh) is implementing the Systems Programming and the Real-Time annex. Groups at Texas A&M University (Richard Volz, Ron Theriaux, Gary Smith) and ENST, France (Yvon Kermarrec, Laurent Pautet) are designing and implementing the Distributed Systems annex.

Ben Brosgol, Dave Emery, and Robert Eachus at Mitre are developing the Information Systems Annex. Jon Squire at Westinghouse Corp. and Ken Dritz at Argonne National Laboratories have contributed several modules of the Numerics Annex. Jean-Pierre Rosen is developing the real-time annex for OS/2. Doug Rupp at the University of Washington is maintaining the DOS and Windows NT ports of GNAT as well as writing Ada bindings to certain Windows NT libraries.

Franco Gasperoni and Patrick Bazire, at ENST Paris, have implemented the GNAT cross-reference facility and are developing a fast recompilation (intelligent MAKE) for the system. Paul Hilfinger at UC Berkeley is enhancing gdb to give it the functionality described in section 9.2 above. Bill Yow has contributed bindings to the OS/2 presentation manager. We want to thank them all, as well as all users whose reports and whose patience with a rapidly evolving system have been a constant source of encouragement.

The GNAT team at New York University, whose work is summarized here, includes: Bernard Banner, Cyrille Comar, Robert Dewar, Sam Figueroa, Richard Kenner, Gail Schenker Morgulis, Brett Porter, Ed Schonberg, and Chien-Chang Sheng. Past members of the project are Bruno Leclerc and Cyril Crozes.

Special thanks to Chris Anderson, head of the Ada 9X project, for her constant support and encouragement.

References

[ADA83] Reference Manual for the Ada Programming Language, ANSI/MIL STD 1815A, AJPO, 1983

[ADA94] Tucker Taft et al, Programming Language Ada, Language and Standard Libraries, Draft 5.0 (proposed ISO/ANSI standard), IR-MA-1363-4, Intermetrics Inc., June 1994

[ADAR94] Ben Brosgol, Robert Eachus and Dave Emery, Information Systems Development in Ada. In *Proceedings of WAdaS '94*, 1994.

[ASIS91] J.B. Bladen et al, Ada Semantic Interface Specification (ASIS). In *Proceedings of Tri-Ada '91*, San Jose, California, 1991

[DIANA83] G.Goos, W.A.Wulf, A.Evans, Jr. and K.J.Butlet. *DIANA - An Intermediate Language for Ada,* Lecture Notes in Computer Science, number 161, Springer-Verlag, 1983

[FAST94] Franco Gasperoni and Patrick Bazire, Smart Recompilation and the GNAT Compiler. In *Proceedings of Tri-Ada '94,* Baltimore, Maryland, 1994 (this volume)

[GCC94a] Richard Stallman, *Using and Porting GNU CC*, Free Software Foundation, 1994

[GCC94b] Richard Kenner. Integrating GNAT into GCC. In *Proceedings of Tri-Ada '94,* Baltimore, Maryland, 1994 (this volume)

[GPL91] GNU General Public License, Free Software Foundation, 1991

[LGPL91] GNU Library General Public License, Free Software Foundation, 1991

[LIB94] Robert B.K. Dewar. The GNAT Model of Compilation. In *Proceedings of Tri-Ada '94,* Baltimore, Maryland, 1994 (this volume)

[PLOEDEREDER92] Erhard Ploedereder. "How to Program in Ada 9X", *ACM Ada Letters, Volume XII*, *Number 6*, (Nov/Dec 1992)

[RTS94] E.W. Giering, F. Mueller and T.P.Baker. Features of the GNU Ada Runtime Library. In *Proceedings of Tri-Ada '94,* Baltimore, Maryland, 1994 (this volume)

[TAG94] Cyrille Comar and Brett Porter. The GNAT Implementation of Tagged Types. In *Proceedings of Tri-Ada '94,* Baltimore, Maryland, 1994 (this volume)