Experiences with Ada Code Generation

Benjamin G. Zorn

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California at Berkeley

ABSTRACT

This paper describes the implementation of an efficient runtime representation for the Ada programming language. This runtime system addresses issues of type representation, package representation, and stack frame organization but does not implement Ada tasking. The structure of the runtime system is discussed in detail. In implementing this system, considerable experience was gained using two intermediate representations of Ada: 1) a high level intermediate form designed for Ada (DIANA), and 2) a low level intermediate form used by the portable C compiler. The utility of both representations is assessed.

December 5, 1984

Sponsored by Defense Advance Research Projects Agency (DOD)
Arpa Order No. 4031
Monitored by Naval Electronic System Command under
Contract No. N00039-82-C-0235.

CHAPTER 1

Introduction

The research described in this paper represents part of an effort at the University of California at Berkeley to study implementation techniques for Ada[1]. The project took advantage of available source code from an Ada compiler donated to Berkeley by AT&T. This compiler, known as the Ada Breadboard Compiler (ABC) was designed as a tool to provide a bootstrap for the development of a production Ada compiler at AT&T [Wet82]. The work at Berkeley was primarily in two areas. One investigation involved implementing alternative representations for the DIANA abstract data type, used as the intermediate representation for the Ada programs [Mur84]. The second area of investigation, described here, involved implementing an efficient runtime representation of Ada programs. Both the form and effectiveness of the representation are described, and comments on the experiences we had implementing the representation are included. The compiler resulting from this work will hereafter be called the Berkeley Ada Compiler (BAC).

This paper is organized in the following way. Chapter 1 is a general introduction. Chapters 2 and 3 recount our implementation experiences with representations of Ada. Chapter 4 describes the BAC runtime representation. Chapter 5 summarizes conclusions reached during this project.

1.1. Overview of What was Implemented

Compilers have traditionally been organized into two phases referred to as the front and back ends. Syntactic and static semantic analyses take place in the front end, which generally produces as output some intermediate form of the program such as triples, quads, or trees. The back end has traditionally taken the output of the front end and produced the target code, which is typically assembler or machine language. Ideally, the front end can be machine independent, and the back end can be language independent.

Modern languages have stretched this paradigm because they are so complex. Furthermore, environment tools have created another use for the intermediate representation of the program. Most Ada implementations have three distinct phases. The front end remains machine independent and creates a high level intermediate form that can be used both by the compiler and other tools in the environment including debuggers, editors and pretty printers. A *middle end* takes this high level intermediate form and produces a more primitive intermediate form as its output. The back end again takes the primitive form and generates the target language. The

[1] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

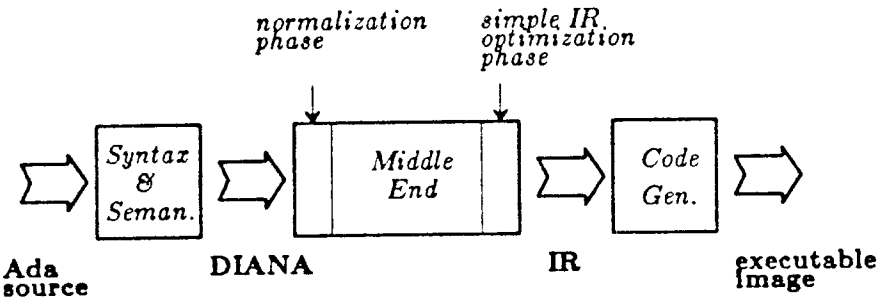

Figure 1.1: Block Diagram of the Berkeley Ada Compiler

middle end, which must manipulate both the high and low level forms, may well be both language and machine ***dependent*** and therefore possibly represents the most time consuming part of retargeting Ada compilers. Figure 1.1 presents a high level diagram of the BAC design.

Because the available resources were limited, and because Ada contains a large number of features, we decided from the outset not to implement features we felt would not be of interest in our research. The three features we ***chose not to implement*** were fixed-point types, derived types, and compilation subunits. As it turned out, the features we ***did implement*** were:

- A full Pascal subset (all Pascal control structures and data types).
- Library and inner packages.
- Static, dynamic, constrained and unconstrained arrays.
- Constrained, unconstrained and variant records.
- All forms of array aggregates and record aggregates.
- Array slices.
- All Ada built-in operators including "&".
- Parameter passing of all data types and function return values of all data types.
- Exception handlers and the **raise** statement[2].
- Constrained and unconstrained access types.
- All scalar attributes and all array attributes.

[2] Constraint checking code was not finished.

- The Interface and Named Interface[3] pragmas.

The features that we were *unable to implement for lack of time* were: tasking, generic packages, default subprogram parameters, many pragmas and many attributes. Also, the I/O packages defined in Chapter 14 of the Ada Language Reference Manual [DoD83] were not provided, although we created the Named Interface to facilitate adding them. Representation specification and many of the implementation dependent features described in chapter 13 of the LRM were also not implemented. Even though we say features have been implemented, none of the features has been exhaustively tested. No substantial user community for the compiler developed because of its unacceptable slowness. The performance of the compiler is discussed in Chapter 5.

1.2. Introduction to DIANA

Chapter 2 describes the experience we had with the high level representation DIANA [GWE83], which is a proposed standard intermediate representation for Ada programs. The DIANA structure is intended to contain information that will be usable by Ada back ends (which include what is called the middle end). DIANA is an abstract data type that allows any number of implementations by abstracting out the interface between the implementation and the user. Although technically the DIANA form is an abstract data type, it most resembles a graph whose principle arcs form a tree. The middle end of the BAC traverses the DIANA representation of the Ada program and generates a tree, the IR of the portable C compiler. The DIANA representation is a complex one, containing definitions for 202 kinds of nodes and 177 attributes, which are the names of fields within nodes.

Overall our experience using DIANA was an unpleasant one. The representation's complexity and poor design hampered the development of the compiler middle end. Furthermore, lack of adequate documentation required much about the structure of DIANA to be learned through trial and error. Despite the criticism of DIANA in this report, there is evidence to suggest that production quality compilers based on DIANA are possible. Verdix Corporation claims to have an Ada front end based on DIANA that compiles code at 1500 lines per minute on a VAX-11/780. If their claim is true, this performance proves that DIANA is not solely responsible for the current low quality of Ada implementations.

1.3. Introduction to the Portable C Compiler IR

The low level form chosen for use in the BAC is the only intermediate form used in the portable C compiler. There were several reasons we chose to use the IR. One reason was that the Berkeley CS community has had a lot of experience with this representation. Three of the high level languages available in the Berkeley Unix[4]

---

[3] Named Interface was a locally defined pragma to allow different external procedures to implement subprograms with overloaded names.

System (4.2 BSD): C, Fortran 77, and Pascal, use the IR as their intermediate form. Consequently, informative documentation on the intermediate form is available [Kes83]. A second reason for using the IR was that another group at Berkeley is implementing table driven code generators [Hen84]. Because their work is based on the same IR, there is hope that their tools can be used to retarget the BAC.

As with DIANA, the interface between the middle end and the IR was abstracted. This abstraction was useful when the need to reimplement the IR arose, as it did several times. Instead of having to modify 20,000 lines of middle end code at each reimplementation, only 1500 lines of IR code needed to be changed. A more complete discussion of our experiences with the IR is included in Chapter 3.

1.4. The Berkeley Ada Runtime Representation

The main purpose of re-implementing the middle end of the AT&T Breadboard compiler was to study the possibility of an efficient runtime representation of Ada, especially with respect to types. Because Ada requires runtime constraint checking and allows unconstrained array and record types, type information at runtime must be available. The BAC implementation minimizes this representation by sharing constant type descriptors among objects of the same constrained type. Package representation is another issue addressed elegantly in the BAC. By avoiding a display mechanism and using static links, package references can be done in a conceptually simple and efficient manner.

Chapter 4 describes in detail the BAC representation of the stack frame, package frames, runtime types, subprogram parameters and function return values. The BAC implementation of record and array aggregates is also discussed. In Chapter 5, the performance of the BAC implementation is discussed along with conclusions drawn from the experiences of implementing the system.

---

[4] Unix is a trademark of AT&T.

CHAPTER 2

Experiences with DIANA

2.1. Overview

DIANA (an acronym for Descriptive Intermediate Attributed Notation for Ada) is an abstract data type suggested to be a standard intermediate representation for Ada implementations. DIANA serves as the connection between the front and back end of the compiler and as a standard representation usable by environment tools such as language-based editors, cross-reference generators, debuggers, and pretty printers. This chapter describes the suitability of DIANA for these purposes from the perspective of a DIANA user. The first part of the chapter discusses general deficiencies in the DIANA design. Next, specific examples of inefficiencies in the DIANA representation are given, and DIANA normalizations implemented by the BAC are described. Finally, important facilities poorly provided by DIANA are outlined.

2.2. DIANA Design Requirements

To qualify as an intermediate representation of Ada usable by environment tools, DIANA had to fill the following requirements [GWE83 p. 9]:

(1) It had to be an abstract data type that would contain all the information determined by syntactic and static semantic analysis.

(2) It had to retain enough information that the source could be reconstructed verbatim[1].

(3) It had to be time and space efficient. Space efficiency includes the efficiency of the external representation which must be maintained across compilations without using too much file space.

(4) It had to provide back end implementors with a representation that was easily understandable and manipulate.

The last two ***requirements*** appears to be an implementation considerations and not design goals. But one must consider that DIANA is intended to be a standard representation and if it is designed to aid the development of Ada compilers and tools, it must not be too inefficient to use for these purposes. The fact that the last two goals were *not* the most important goals in the DIANA design had a negative impact

[1] Actually, the greatest aspect of source code reconstructibility left to the discretion of the implementation was whether or not to retain comments from the source code in the DIANA representation.

on the result.

2.2.1. The Impact of Source Reproducibility

The most questionable DIANA design goal was reconstructibility of the Ada source. There is a great deal of importance placed on this goal, even though it conflicts directly with the goals of usability and space efficiency. Which is more important, source reproducibility or space efficiency and compiler usability? One can resolve this question by examining the frequency of use and importance of the particular tools that require exact source reproducibility. Because the tools in question: editors, pretty printers, debuggers, etc., can provide a meaningful representation of a program to the user independent of the original source (i.e. print the program out in some standard format), the marginal utility of maintaining all the additional information so the user can see exactly what he typed seems questionable.

The most frequent thing one does with a piece of software is compile it. Software that works perfectly is repeatedly compiled because interfaces change. In Ada, if a variable is added to a package specification, then all program units that have a dependency on that specification need to be recompiled, even if they never access that variable[2]. Ideally, interfaces are frozen early in a project and never change thereafter. Unfortunately, this rarely happens. And because interfaces change, recompilation happens again and again to entire working software systems.[3] It is clear that space efficiency and compiler usability should be high priorities for designers of any intermediate language representation.

Ignoring the source reproducibility requirement would have drramatically changed the structure of DIANA. Specific changes are discussed throughout this chapter. If a user desired it, source information could have been retained by augmenting the DIANA. The most awkward problem here is that two different intermediate representations would exist; however, the current alternative is even less desirable.

2.2.2. The Basis for the DIANA Design

The DIANA design team chose to base DIANA on the Ada Formal Definition (AFD) [Hon80]. The AFD was an attempt to formalize the static semantics of preliminary Ada [Ich79]. The DIANA designers chose to base DIANA on the AFD so that the intermediate representation would have meaning and consistency that would not need to be provided by the DIANA design team independently. Unfortunately, since the AFD design was discontinued before the final Ada standard [DoD83] was issued, the AFD does not specify the correct semantics for all Ada language features, and hence formal basis for DIANA is incomplete.

---

[2] It is possible for a compiler to avoid recompilation through a non-trivial analysis, but this capability is not required.

[3] Our compiler was organized into three distinct passes. They all depended on the DIANA interface. Whenever the DIANA interface changed, perhaps because a code generation attribute was added, each of the passes had to be recompiled.

Ideally, the DIANA representation should be a minimal representation that preserved the analysis of the front end. The creation of this representation requires two transformations to the source, first from the source to the abstract syntax tree (AST) and then from the AST to DIANA by means of decoration[4]. The AFD defines the structure of the AST, and also defines the semantics of decorating it. Thus the AFD provides the DIANA designers with answers to questions of the meaning (what does a particular DIANA structure represent), and consistency (is a particular representation meaningful). The biggest problem with basing DIANA on the AFD is that the AFD was not designed for efficiency of representation. Furthermore, it seems unwise to create a de facto standard representation based on an out-of-date specification upon which no successful compilers have been written. It seems it would have been much more prudent to delay the design of DIANA until a number of successful Ada implementations had been completed.

2.3. The Size of the BAC DIANA Implementation

The BAC DIANA implementation is based on the DIANA for the Ada Breadboard Compiler designed at AT&T [Qui82]. This implementation was modified at Berkeley to minimize the size of the representation. Murphy calculated that in the BAC the DIANA structure for a given Ada program would be 23 times the size of the source text [Mur84 p. 15]. Even with rough calculations, the size of the DIANA structure can be seen to be large. Assume that the DIANA representation is a binary tree (a conservative estimate, since most DIANA nodes have more than two children). Furthermore, assume that the tokens in the source become leaves of the tree. Based upon these assumptions, there are twice as many nodes in a program's DIANA representation as there are tokens in the source. Murphy calculated that the average node size in the BAC DIANA implementation was 35 bytes [Mur84 p. 15]. If we assume two DIANA nodes per source token, the ratio is 70 bytes of DIANA structure to one source token. If the average source token size is 4 bytes, a source to DIANA structure expansion factor close to 20 (70/4) is believable.

Just the information to represent source position takes up large amounts of space. In our implementation, every node has a source position, which is represented as two 2 byte integers (row number and column number). Assuming again that the average token size is 4 bytes and that there are two DIANA nodes for every source token, the space devoted to maintaining source position information takes up twice as much space as the source itself.

The experience with the BAC implementation of DIANA indicates that ***any*** DIANA implementation will be an inefficient representation of the Ada source. Consider the representation of lists (SEQ_TYPE) in DIANA. Figure 2.1a illustrates the DIANA representation for a list of identifiers in the declaration part of a subprogram.

---

[4] The structure of the AST and DIANA are virtually the same. The transformation involves filling in attributes (decorating) the nodes in the AST.

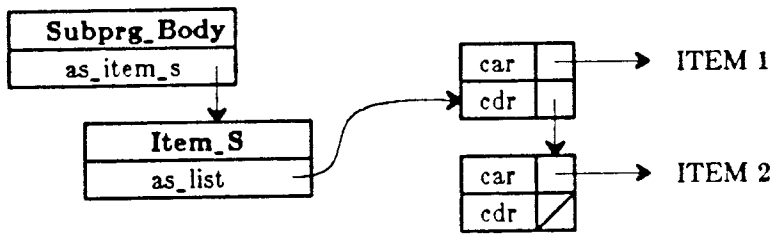

Figure 2.1a: DIANA Representation of a Declaration List

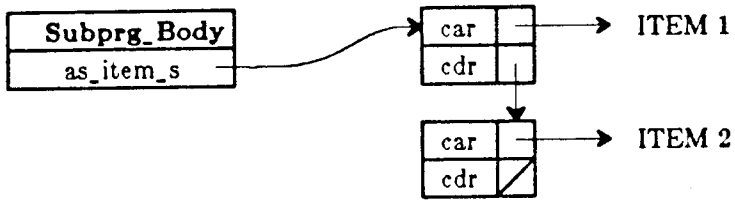

Figure 2.1b: Simplified DIANA for a Declaration List

In these figures, the darker first name in each node represents the kind of the node. Fields underneath this represent the various attributes of the node, some of which are pointers. The car—cdr structure suggests a familiar implementation of lists. In DIANA, lists are defined as an abstract data type, and so the implementation suggested by these diagrams is only one possible implementation of many. In the nodes represented, many of the attributes are not shown (the largest node has 14 attributes) to simplify the figures.

Notice the Item_S node in Figure 2.1a. This node has only one attribute, AS_LIST, which points to the list of declarations. There appears to be no purpose in making the Item_S node separate instead of placing the list of identifiers in the Subprg_Body node. Why not have the AS_ITEM_S attribute point directly to the list of identifiers as is represented in Figure 2.1b? This modification would avoid the overhead of the Item_S node, and all the extra pointers, complexity, etc. In DIANA, every node whose name ends in _S is just a dummy node that has one attribute, AS_LIST. This redundancy is clearly non-optimal.

2.4. Useful Normalizations of DIANA

The source reconstructibility goal requires that much of a DIANA structure remain unnormalized. If the intermediate form represented a canonical form of the Ada program, the job of retargeting Ada compilers would be much easier because back

end writers would only need to understand and manipulate a single representation of the Ada source. In an unnormalized DIANA structure, there can exist many different forms that represent the same semantic meaning, all of which must be individually recognized and normalized by the back end. Normalization of the DIANA structure would also reduce the complexity of the representation.

Example 2.1a and 2.1b contain fragments of Ada source that are semantically equivalent.

```
x : integer;
y : integer;
```

Example 2.1a: Declaration Form One

```
x, y : integer;
```

Example 2.1b: Declaration Form Two

Figure 2.2a illustrates the DIANA representation of Example 2.1a. Note the structure replication for each variable declaration. The DIANA representation also introduces an redundant node for every declaration, which is a header telling the type of the following list. The Var nodes in Figures 2.2a and 2.2b are extraneous. In Figure 2.2b the DIANA representation of Example 2.1b is presented.

A normalized representation for Examples 2.1a and 2.1b is illustrated in Figure 2.3. In it the Var and Id_S nodes have been completely eliminated. Furthermore, nodes named Exception, Constant, Subtype, Type, In, Out, In_Out, L_Private, Private, and Number could be eliminated from the DIANA definition. Including the list nodes (ending in _S) and these nodes, more than 25 DIANA node kinds are extraneous in representing results of syntactic and static semantic analysis.

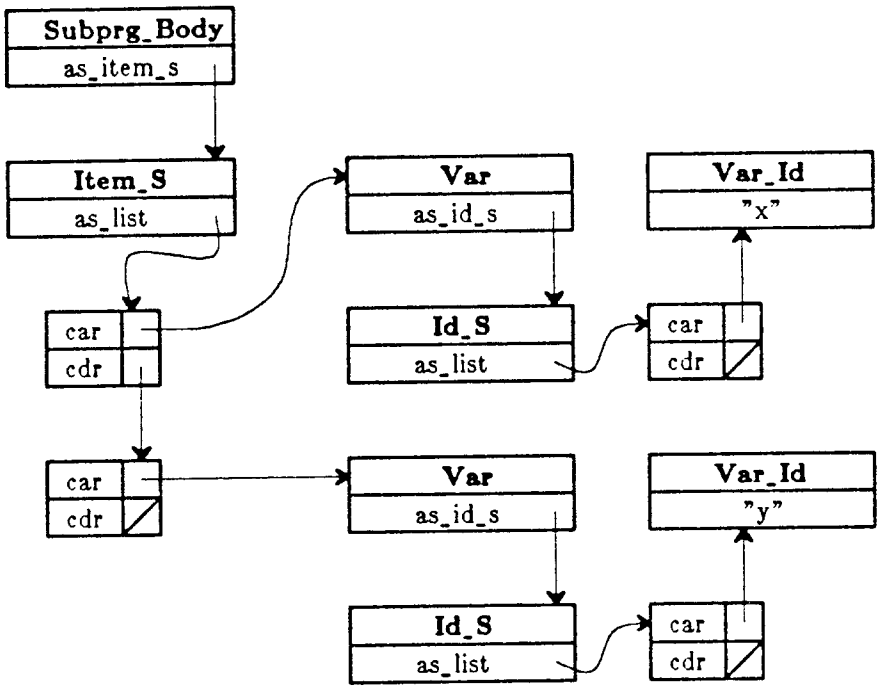

Figure 2.2a: DIANA for Example 2.1a

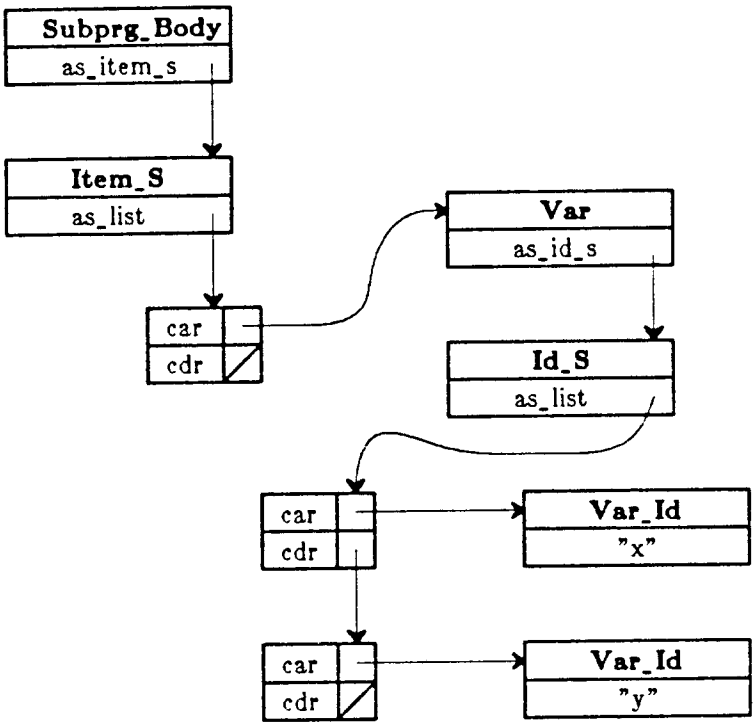

Figure 2.2b: DIANA for Example 2.1b

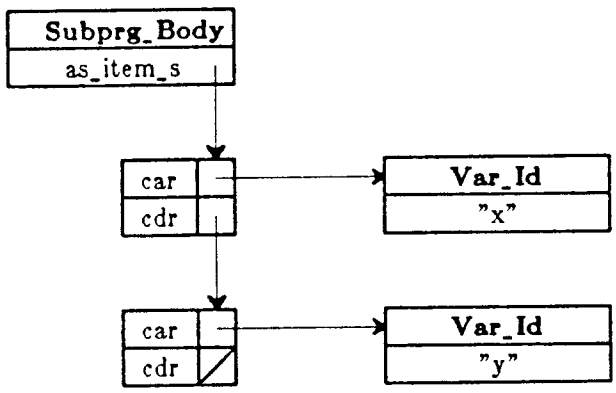

Figure 2.3: Normalized DIANA for Examples 2.1a and 2.1b

2.4.1. Necessary Ada Type Normalizations

In Ada every constrained subtype is represented at runtime by a subtype descriptor. Normalizations are necessary because constrained subtypes are required that are not explicitly represented in the source text. Example 2.2 illustrates this point.

```
x : glorp(1 .. 10);
```

Example 2.2a: Unnormalized Implicit Subtype Declaration

```
subtype $anon1 is glorp(1 .. 10);
x : $anon1;
```

Example 2.2b: Example 2.2a with Subtype Explicit

Anonymous subtype normalization must be done for implicit subtype declarations in all contexts, including array components and record components.

A more difficult and important type normalization involves normalizing array types. Consider Example 2.3. The Ada Reference Manual [DoD83 p. 3-29] specifies that all of these array declarations are semantically equivalent. The DIANA representation does not unify these different forms, even though the front end will calculate the normalized array forms to do semantic analysis. Because the DIANA representation is not allowed to contain the normalized forms, the back end must recompute the information.

2.4.2. Summary of Normalizations Implemented

In the BAC implementation, normalizations were done to the DIANA structure between the front end and the middle end (see Figure 1.1). The BAC normalization phase normalized all array types and implicit subtypes. All occurrences of the attribute 'RANGE were changed to the range 'FIRST .. 'LAST. Record aggregates, which in the DIANA structure appear exactly as they do in the text, were normalized to always appear in positional order. Similarly, named parameter lists were normalized into positional order to make parameter passing easier. These normalizations made the BAC middle end easier to implement.

```
type A is array (1 .. r1, 1 .. r2) of Comp_Type(1 .. 10);
an_A : A;
```

Example 2.3a: First Form of Array Declaration

```
an_A : array (1 .. r1, 1 .. r2) of Comp_Type(1 .. 10);
```

Example 2.3b: Second Form of Array Declaration

```
subtype $anon1 is integer range 1 .. r1;
subtype $anon2 is integer range 1 .. r2;
subtype $anon3 is Comp_Type(1 .. 10);

type $anon4 is array ($anon1 range <>, $anon2 range <>)
                of $anon3;
subtype $anon5 is $anon4($anon1, $anon2);

an_A : $anon5;              -- $anon5 is called A in 2.3a
```

Example 2.3c: Normalized Form of Array Declaration

2.5. DIANA's Lack of Useful Information

Another problem with the DIANA representation is that is does not contain information necessarily present during semantic analysis. Example 2.4 illustrates this point.

```
type A is array (integer range <>) of integer;
x : constant A := (1, 2, 3);
```

Example 2.4: Important Type Information not Provided by DIANA

The declaration of x is legal. The initial value of the constant x provides the constraint on the array type A. Thus, the subtype of x, A(1 .. 3), implicit. However, the DIANA structure for the definition of x shows no that the type of x is A, and shows no constraint even though every legal array variable in Ada *must* have a constraint. To do correct semantic analysis, the actual subtype of x has to be

computed, but the DIANA structure does not preserve this information.

A well designed symbol table would have also been helpful to back end writers. As Example 2.4 demonstrates, the DIANA contains source structure information about variables instead of semantic information. Information obtained from semantic analysis (like an object's normalized type) is not summarized in any way. Furthermore, there is no suggested symbol table organizational (like a hash table) that would make semantic analysis more straightforward. Since DIANA is an entirely abstract interface, this could be provided in the implementation, but such an implementation would be quite sophisticated because it requires the access method of a particular node to be sensitive to the node kind. The ABC implementation did not take advantage of this abstraction because they wanted to use a uniform pointer implementation of DIANA to gain efficiency [Wet82]. The result was that the symbol table used for semantic checking in the ABC implementation is separate from the symbol table provided in standard DIANA (although they did define their new symbol table as an extension of DIANA, in the same formalism used to define DIANA, Lamb's Interface Description Language [Lam83]).

Quinn and Wetherell also defined Artemis, a DIANA-like intermediate representation for Ada, that includes symbol table organization explicitly, and is believed to be a more compact representation. [QuW83]

Runtime type representation is another implementation issue avoided by the DIANA design. Because the DIANA designers did not want to advocate a particular runtime type representation, DIANA does not contain offset attributes. The argument against having an explicit symbol table organization and providing code generation support (e.g. code attributes) is that DIANA is intended to be as simple as possible and implementations can provide their own additional features. However, this argument fails on two counts: first, if there is a need for a particular feature (like a conventional symbol table) then everyone will have to provide one, and they will all be different (and possibly redundant, as is the symbol table in the ABC); and second, if DIANA is a stripped down intermediate form, then why is it so complex, especially considering that much of the omitted support must be provided anyway. The Artemis design provides attributes for elements of the symbol table that support code generation, optimization, and debugging, and the representation is less complex.

CHAPTER 3

Experiences Representing Ada with the C IR

3.1. Overview

The IR is a low level representation of C expressions. Expressions are represented in a tree structure, and there are limited facilities for control in the form of sequence operators. The IR contains constructs for most of the C operators, and also allows explicit referencing of registers. This chapter describes the experiences we had using the C IR as a low level representation for Ada. A possible alternative representation is discussed, and the two representations are compared. Also, unpredicted benefits of using the IR are presented. Finally, the actual suitability of the IR for representing Ada is described.

3.2. Reasons for Choosing the IR Representation

The most important reasons for selecting the IR as the low level representation for Ada were given in the introduction. The only other candidate considered was the language C itself. The original Ada Breadboard Compiler generated C source code. There are several advantages to using C as the target language. First, since the compiler itself is written in C, debugging facilities available for compiler development automatically are available for debugging the executable code of the compiler. Second, because C is textual, it is easy to understand the code the compiler is generating, and easier to find and fix bugs. Finally, because C and Ada have common control structures, using C implies that control structure implementation is trivial.

There are also disadvantages to using C as the target language. A primary drawback is the full C compilation required after the Ada compilation. With BAC compilation speeds agonizingly slow anyway, this extra time just makes a bad thing worse. Nevertheless, if using C would improve development time of the middle end dramatically, such a trade-off might have been acceptable. It has turned out that the IR represents a sufficiently large part of C that the ease of generating C source is balanced by the availability of machine level operations in the IR. These operations include accessing registers (the frame and stack pointers) and generating non-C procedure calls (VAX *jsb*). The greatest disadvantage of using the IR it that control structure implementation (especially the case statement) is more difficult. Implementing control structures is a minor part of the overall difficulty of the project however. The ease of understanding C made little difference in our implementation because little time was spent debugging problems in the IR representation.

3.3. Other Benefits from Using the IR

An unforeseen benefit of using the IR representation was that the representation could be easily optimized in simple ways. The IR representation explicitly contains all addressing. Certain stylized forms of the IR, such as referencing with a constant displacement off a register, can be converted directly into VAX addressing modes. Also, constant folding and identity simplification is straightforward. The IR optimizer is a small program (about 450 lines of C) that fits between the middle end and the code generator in the BAC (see Figure 1.1). Overall, the optimizations carried out by this code reduced the size of the IR representation more than one third in many cases.

Another advantage of the IR was that it allowed a fairly clean interface with the middle end. The interface enforced a pre-order tree construction discipline with strict stack operations. Currently, only 5 functions[1] form the entire interface between the middle end and the IR implementation. Using an abstracted interface allowed reimplementation of the IR representation on two occasions. The initial implementation of the IR was a memory resident tree structure that was dumped to a file when it was completely constructed. In an effort to speed up the middle end this representation was discarded in favor of forcing the user to generate the nodes in a pre-order fashion, and dumping each node to a file as soon as it is created. To achieve this goal, the simple stack operation interface was implemented on top of the original tree manipulation interface. This strategy avoided the overhead of the memory allocation needed to represent the tree internally, with the disadvantage that none of the tree nodes were available after they were created. Later, when optimization of the expression trees was desired, another representation became necessary. This time the IR trees for individual expressions were maintained in memory until the expression was completed, and then the optimizer would be run on it before it was dumped to a file. This organization has proved to be the most successful of the three.

3.4. Suitability of the IR for Representing Ada Programs

One Berkeley researcher commented that he saw representing Ada in the IR as akin to trying to force a large object into a small space using a funnel. One might wonder how well the IR fares in representing Ada. Our experience has shown that the IR is quite suitable for this task. Because virtually any instruction sequence desired can be created by combining the right IR nodes, the IR provides the flexibility of an assembler representation. However, because it is somewhat strongly typed, provides many operators as well as simple calling structures, and handles temporary register allocation for the user, the IR makes the job of code generation considerably easier than generating assembler directly. Finally, because the IR is relatively machine independent for a certain class of architectures (like the VAX, M68000, etc.), using the IR allows implementors to retarget Ada compilers with less effort.

---

[1] The functions are: push_leaf, push_op, print_expr, ir_init, and push_dummy_leaf. The names suggest their functions. Push_dummy_leaf is needed because a particular operator (FEXPR) always expects two operands, and sometimes they are both not necessary.

We did have some problems with the IR representation. One fault is the inflexibility of the calling convention used. Because function calls are embedded in expression trees, the IR user has to be satisfied with the calling convention that the code generator chooses to use. In the case of the code generator we use on the VAX, procedure and function calls are done using ***calls***. While this allows different languages to do inter-language procedure calls because they have the same calling conventions, the execution speed of the ***calls*** instruction is sufficiently slow that another calling convention would be preferred.

Another problem with the IR is that its low level prohibits certain optimizations. These optimizations are impossible for several reasons. First, stack addresses are represented explicitly in the IR, making dead variable elimination a much more difficult problem. This particular optimization is of special interest because the runtime type system in the BAC generates unused type descriptors that could potentially be eliminated. Also, the IR has a node called FTEXT, which basically is a pass-through node that generates whatever assembler text the user wants to place in it. The existence of this node makes any sophisticated optimizations on the IR virtually impossible, because the representation becomes more machine dependent, and without knowing what code the FTEXT nodes generate, the optimizer must make extremely conservative assumptions. It is clear that the IR was designed to be a portable representation of C without the potential for sophisticated optimization.

A final criticism of the IR has to do with its current status. While it defines nodes labeled SWITCH, FLOAT, and STRING, they are all unimplemented, and produce errors when used. The intent of these nodes seems clear enough, and though implementing the capabilities they imply is trivial, it is largely machine dependent, and reduces the portability of the representation.

CHAPTER 4

Berkeley Ada Compiler Runtime Organization

4.1. Overview

This chapter contains the specific details about the organization of the runtime environment in the BAC implementation. The design rationale is given as well as many specific examples to make the implementation understandable. First, the structure of the runtime stack is given along with the reasons for the subroutine linkage conventions. Next, the representation of the library, inner, and generic packages is discussed. Following the package description is the description of the runtime type representation, to which the majority of the chapter is devoted. Scalar, real, array and record types are discussed. The decisions involved in the designing the structure of record types are then presented. The end of the chapter addresses specific implementation issues including representation of parameters, function return values, slices, array aggregates and record aggregates.

4.2. Contents of a Stack Frame

In the BAC runtime system implementation, we chose to have 4 storage allocation pools. They are:

(1) The runtime stack, from which subroutine local data is allocated, and in which subroutine linkage takes place. This is the only pool with an interesting structure.

(2) The statically allocated data space, where statically sized objects (such as string literals, floating point literals, and, library package data) are allocated.

(3) The heap, a global pool of data used to allocate dynamic objects.

(4) The temporary stack, used to allocate temporary results that are dynamically sized.

The BAC uses a stack frame structure compatible with the VAX *calls* instruction [DEC81]. This allows inter-language compilation between Ada, C, Pascal, and Fortran 77. The overall structure of the stack frame is shown in Figure 4.1. In invoking an Ada subprogram, the calling subprogram pushes the appropriate arguments onto the runtime stack, along with the number of arguments. The *calls* instruction saves the state of the calling subprogram (the old frame pointer, argument pointer, registers, etc.) on the stack. Then the called subroutine is instantiated, and the space necessary for static local data is allocated. Finally, during the elaboration of the declarations, space necessary for dynamic local data is allocated.

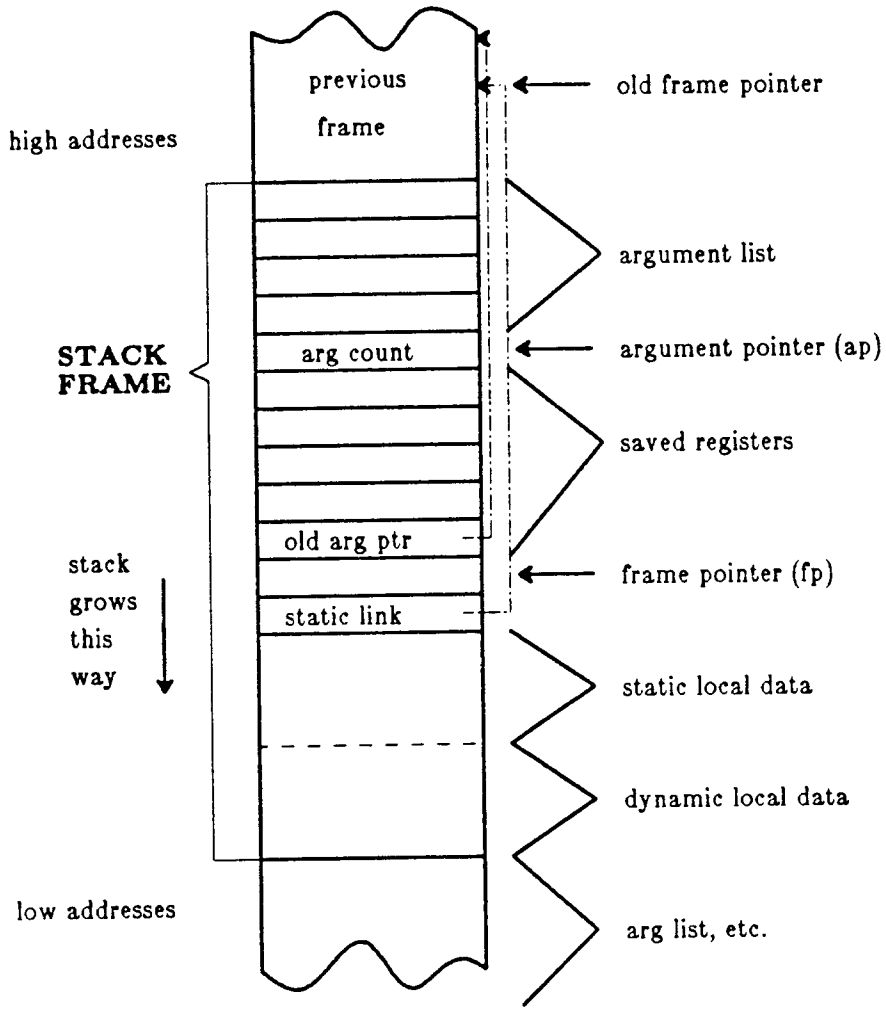

Figure 4.1: Contents of a Stack Frame

The objects allocated for a particular instantiation of a subprogram include local constants, variables, type descriptors, exception handling information, and compiler allocated temporaries. The exact structure of data and descriptor objects is described below. Compiler allocated temporaries include case selectors, loop bounds, and things that Ada semantics require must be computed once, but may be referenced multiple times.

4.3. Representing Dynamic Objects in the Runtime Stack

The overall design strategy for representing dynamically sized objects is illustrated well by the structure of a runtime stack frame. Consider Example 4.1.

```
procedure dyn is
    length : integer := read_in_some_length_procedure;
    dyn_arr1 : array (1 .. length) of integer;
    dyn_arr2 : array (-length .. length) of integer;
begin
    ...etc.
end;
```

Example 4.1: Sample Procedure containing a Dynamic Array

All objects are partitioned into static objects and dynamic objects. In this case, the integer length, and the descriptors for the variables dyn_arr1 and dyn_arr2 are static objects. Descriptors are a known fixed size. Static objects have a size known at compile time, and can be assigned a fixed offset in the stack frame. Dynamic objects (the data for variables dyn_arr1 and dyn_arr2) cannot have a fixed offset because their size varies between instantiations of the procedure dyn.[1] For each dynamic object represented, a word is allocated in the fixed part of the stack frame that contains an offset into the dynamic part of the stack frame. This offset provides the location of the particular dynamic object for a given instantiation. Figure 4.2 diagrams the actual structure of the stack frame for the procedure dyn. A similar strategy is used to represent dynamic records.

4.4. Subroutine Linkage in the Runtime Stack

The subroutine linkage convention used in the BAC matches that of the VAX *calls* instruction. The parameters are first pushed onto the stack. Next, the *calls* instruction saves the return address, old frame pointer, argument pointer, and the registers specified by the register mask. The old frame pointer is the dynamic link that provides access to the caller's stack frame. The *ret* instruction restores all the saved registers, and deallocates the current stack frame. The only linkage feature of Ada not provided by the *calls* instruction is the static link, the pointer to the textually enclosing frame of a subprogram to allow up-level access to variables in outer scopes.

The static link in the BAC design is provided by the calling program, and always passed as the first argument in the argument list. Upon entry the static link is saved in a local temporary, and stored in a register as well, to provide fast access to

[1] Actually, the data for variable dyn_var1 always begins at the known last word of the static part of the stack frame, but this optimization only applies to the first dynamic object in a stack frame.

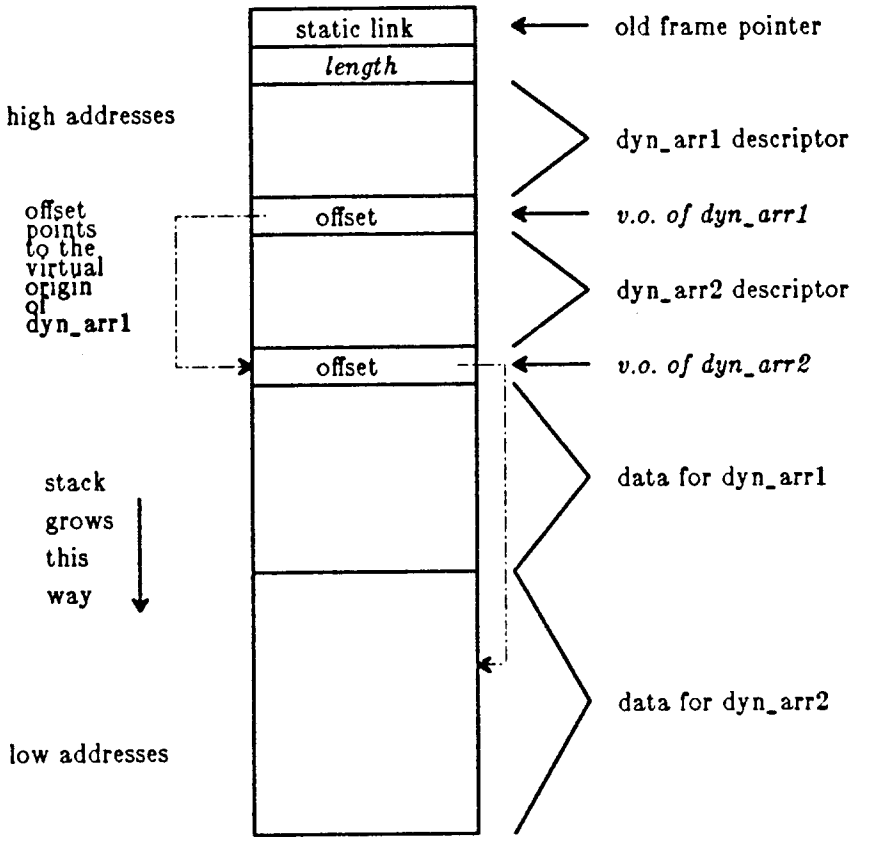

Figure 4.2: Structure of Stack Frame for Example 4.1

variables in the adjacent enclosing scope. The local copy of the static link is used for up-level addressing to variables in scopes outside the adjacent enclosing one. A possible enhancement to this design would place a pointer to the outermost scope in a register permanently, thus allowing much quicker references to the outermost scope. In analyzing 120,000 lines of Pascal programs, Cook and Lee determined that more than 97% of non-local variable references are made to variables in the outermost scope [CoL82].

We decided to use static links rather than a display to reference non-local variables. The greatest advantage of a display is that it allows equal access times to objects in any scope. Static links in the worst case require time proportional to the difference in nesting levels of the current scope and the scope of the object being referenced. We believe that the number of references to outer scopes was sufficiently infrequent to outweigh the disadvantages of displays, discussed below. In addition, by making the scopes most often referenced accessible through registers, almost all of the

overhead from using static links is eliminated.

A major disadvantage of displays is that they require special action to be taken during an exception. In the normal case, a subprogram will restore the old display value before returning. An exception can be handled by a non-local handler (causing a *goto* out of a scope). Because the flow of control is not through the standard return mechanism, action must be taken to restore the display to its correct state. This restoration can be done in the exception handler, but requires the stack to be unwound. Some of the problems involved with stack unwinding in RIGEL[2] are described by Cortopassi [Cor81]. The BAC static link implementation requires no stack unwinding.

Finally, using the display mechanism in conjunction with linkage to package subprograms was unnecessarily complicated. The ABC implementation used displays for up-level referencing, and maintained a separate package display to handle package variable references [Rub82]. Implementing package references using static links is a very simple and elegant process.

4.5. Package Representation

In Ada, packages create a syntactic guard that provides a useful tool for data abstraction. In our implementation, several kinds of packages are of interest:

- Library packages—or outer packages—are packages declared as separate program units. Library packages are accessed using the Ada **with** statement. Since no scopes enclose library packages, static linkage for functions declared in the package is not difficult to provide.
- Inner packages are packages declared within a compilation unit. Functions in inner packages can access objects in scopes that textually enclose the package, and so providing static linkage to these scopes is necessary.
- Generic packages are library or inner packages that are parametrized and allow multiple separate instantiations. The current BAC does not implement generic packages. Sharing code between instantiations of generic packages is an important issue, and generic packages are discussed.

The following sections discuss the location of the data, the environment linkage, and package instantiation for each of these three cases.

4.5.1. Library Package Representation

Because library package data is globally accessible to any compilation unit that includes it using a **with** clause, library package data can be allocated in the static data pool. Our implementation has not addressed the problem of where to put dynamic data that is part of a library package declaration. The best place to put it is

[2] RIGEL is a language developed at Berkeley that features exception handling semantics very similar to those found in Ada.

in the heap where it will be allocated once and never deallocated. This feature remains to be implemented, and differs from the technique of allocating dynamic objects after all the static objects described above.

Another problem is how to provide the static linkage to subprograms in packages called from outside the package. The only static environment needed by top-level functions in a library package is the data in the package specification, and since the location of that data is globally known, the static link in this case is irrelevant.

Library package instantiation (including the elaboration of the package body) takes place prior to the elaboration of any of the program units that use the package, and the instantiation order is determined using a partial dependency graph and a linker provided in the ABC implementation.

4.5.2. Inner Package Representation

Inner packages are declared inside subprograms. In the BAC implementation, storage for the static data of inner packages is allocated in the static part of the stack frame of the enclosing subprogram. Storage for dynamic package data is allocated in the dynamic part of the enclosing stack frame.

Example 4.2 illustrates a typical inner package. Providing the correct environment to subprograms declared in the package, and called from outside the package (like inner.F) is more difficult for inner packages. The strategy used in the BAC is to make the package data nested in the enclosing scope appear exactly like a stack frame. The static link of the ***package pseudo stack frame*** is the enclosing scope of the frame. The runtime representation of the program in Example 4.2 is presented in Figure 4.3. When a subroutine calls inner.F, the linkage is as follows: first the caller knows that inner.F is a package function, and further knows the offset of the package pseudo stack frame within the stack frame of the enclosing procedure (enclose). This known address is pushed as the first argument (the static link) in the call to inner.F. Because the package specification pseudo stack frame is organized as shown in Figure 4.3, the pseudo stack frame looks just like another frame to inner.F. The reference to local_var is up one level; the reference to enclose_var1 is up two levels, following the link from the pseudo stack frame to the frame of enclose; and the reference to the outer_var follows the static link from the frame of enclose to the scope of outer_var.

The only additional thing that needs to be done is that the static nesting levels of subprograms in the package need to start at a number one greater than the static nesting level of the package. The nesting levels have to be changed because the package specification pseudo stack frame is an interposing static level between the frames of package subprograms and the frames of enclosing subprograms.

```
outer_var : integer;                -- visible inside package

procedure enclose is
    enclose_var1 : integer;         -- visible inside package

    package inner is
      local_var : integer;
      function F return integer;
      ...etc.
    end;
    package body inner is
      function F return integer is
      begin
         local_var := 1;            -- reference to header local
         enclose_var1 := 2;         -- reference inside enclosing proc
         outer_var := 3;            -- reference outside enclosing proc
         ...etc.
      end;
      ...etc.
    end;
    enclose_var2 : integer;
begin;
   outer_var := inner.F;            -- call to package function
   ...etc.
end;
```

Example 4.2: Sample Inner Package

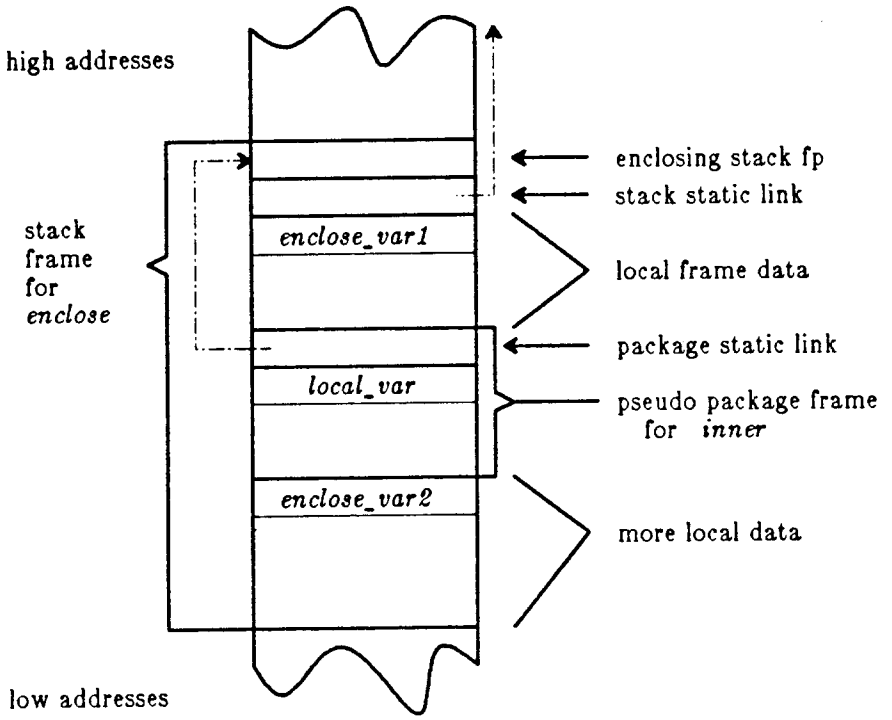

Figure 4.3. Static Linkage for Elaboration of Inner Package Specification

4.5.3. Generic Package Representation

The most obvious implementation of generic packages involves doing the equivalent of a macro expansion for each package instantiation. While this approach is conceptually simple, it has serious drawbacks when a package is instantiated many times. The alternative is for each package instantiation to share code. In this case it is important that all instances of a package use the same code sequence to access package objects. Consider Example 4.3. The issues this example highlights are the location of the instantiated package's data, how package data is accessed, how package procedures are called, and how up-level references from within the package are implemented.

The strategy for allocating the generic package data is identical to the strategy outlined for inner packages; the data is allocated in a pseudo stack frame in the stack frame of the program unit in which the package is instantiated. This strategy is illustrated in Figure 4.4. Note that the static link for the instantiation does not point to the frame enclosing the program unit in which it is instantiated, but instead to the frame enclosing the generic definition.

```
outer_var : integer;                -- visible inside package

generic
package example is
    ...etc.
end;
package body example is
    local_var : integer;

    function F return integer is
    begin
      local_var := 10;    -- reference to package local
      outer_var := 11;    -- reference outside package def
    end;
       ...etc.
begin;
   ...etc.
end;

...etc.

package new_example is new example;
```

Example 4.3: Sample Generic Package

Now consider a call to new_example.F. The static link passed to F must be the address of the pseudo stack frame. Once inside F, a reference to local_variable involves following the static link passed from the stack frame of F to the pseudo stack frame of new_example. A reference to outer_variable involves following the static link from F into the pseudo stack frame for new_example, and then following its static link to the environment in which the generic example package was declared to correctly reference outer_variable. Generic non-procedure parameters in this context are treated simply as package variables, and procedure parameters are handled in the conventional style of passing the procedure's static link and its entry address.

4.6. Runtime Type Representation

Ada requires that type information be represented at runtime. In the BAC implementation this runtime type information is stored in a runtime object called a *descriptor*. There are various kinds of descriptors in the BAC implementation, and they contain information such as range constraints, discriminant values, and offsets

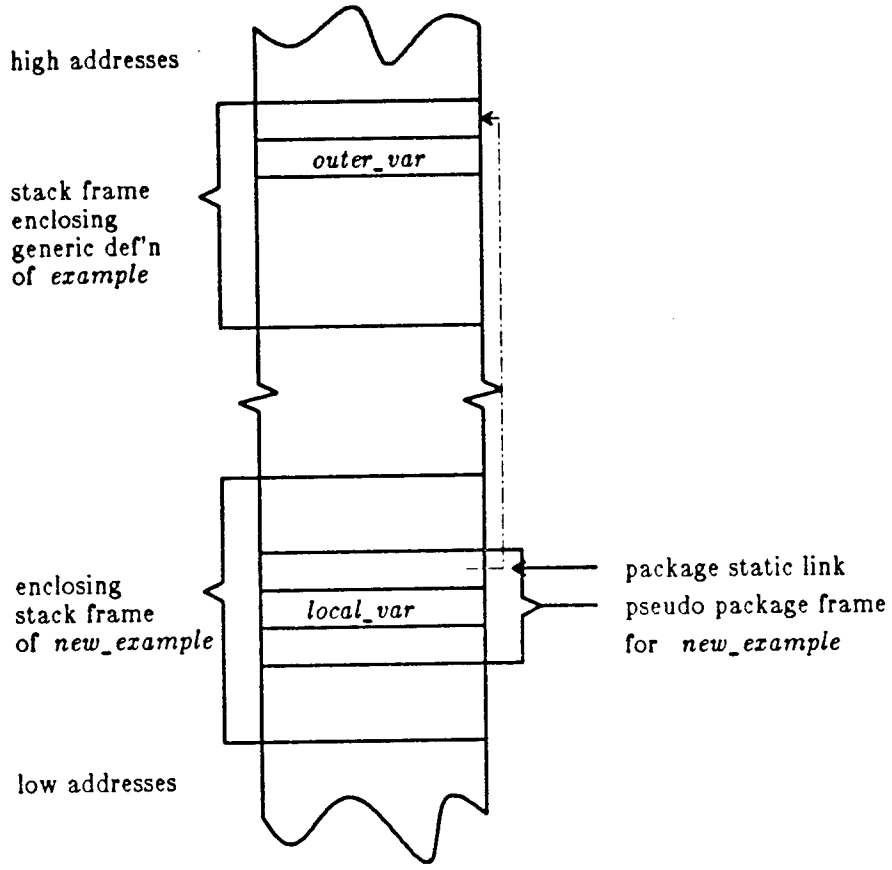

Figure 4.4: Linkage for Instantiated Generic Package in Example 4.3

into data objects. Data objects in the BAC implementation consist of what is normally considered the value of a particular object, such as the elements in an array, or the components of a record. The format of descriptor and data objects will be described in the following sections. Because the BAC implementation did not consider the representation of fixed point types, they are not discussed here.

4.6.1. Consequences of Type Definitions

When a programmer declares a type in the BAC implementation, he conceptually gets more than that. With each type definition three representation types are potentially defined. They are: T_$desc, the descriptor type for type T; T_$data, the proper data type for type T, and T_$full, which is a concatenation of the $desc and $data types for type T. These types are not all meaningful for every type declaration. The various types of Ada objects are as follows:

- A constrained variable or constant of type T is represented by an object of type T_$data.
- A constrained subtype of type T is represented by an object of type T_$desc.
- An unconstrained subtype of type T is not represented at all at runtime.
- The type T is not represented at all at runtime, except in a special case called *subtype records*.
- An unconstrained record type of type T is represented by an object of type T_$full.
- An unconstrained array function return value is represented by a 4 byte pointer to the virtual origin, and an object of type T_$full.
- An access type with a constrained base subtype is represented by a pointer to an object of type T_$data.
- An access type with an unconstrained base type or subtype is represented by a pointer to an object of type T_$full.

In the BAC implementation, there are four kinds of descriptors: scalar, real, array, and record. All descriptors except record descriptors have a fixed format.

4.6.2. Scalar and Real Descriptors and Data

Scalar data (of enumeration and integer types and and subtypes) is represented by 1, 2, or 4 byte quantities, whichever size will allow representation of the entire range. Scalar descriptors are represented by two 4 byte quantities, which contain the lower and upper bound of the range of the subtype.

Similarly, real numbers are represented with either 4 bytes (single precision, or **float digits** 7) or 8 bytes (double precision, or **float digits** 15). The descriptors for constrained real subtypes contain two 8 byte real numbers, representing the lower and upper values for the constraining range. Example 4.4 illustrates a typical Ada program containing scalar and real values, and Figure 4.5 illustrates the stack frame for Example 4.4.

```
procedure scalar_real_example is
    type color is (red, yellow, green, blue, purple, black);
    subtype dark_color is color range blue .. black;
    a_color : constant color := red;
    a_dark_color : dark_color;
    subtype small_range is integer range 1 .. 150;
    an_integer : small_range;
    a_big_integer : integer := 100000;
    subtype real_range is float range 1.0 .. 3.0;
    a_real : real_range;
    a_float : float;
begin
    ...etc.
end
```

Example 4.4: Procedure containing Scalar and Real Types

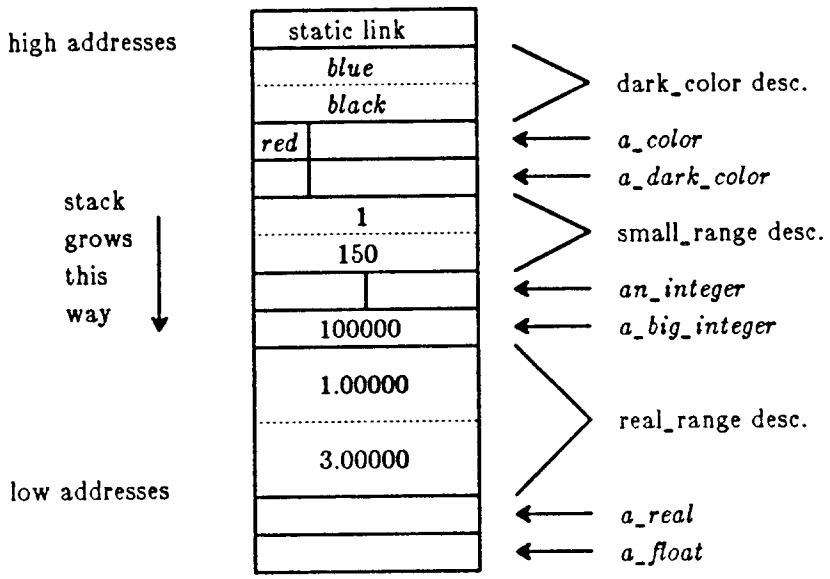

Figure 4.5: Structure of Stack Frame for Example 4.4

4.6.3. Array Descriptors and Data

Array data in the BAC implementation is stored in contiguous space in row-major order. Array descriptors contain exactly three 4 byte numbers for each array dimension. The three numbers are: the multiplier, lower bound and upper bound for each dimension of the array. Static arrays are allocated in the static part of the stack frame. Dynamic arrays are allocated in the dynamic part of the stack frame, and a 4 byte pointer is allocated in the static part of the frame which points to the ***virtual origin*** of the array. In the BAC implementation a pointer to array data points to the virtual origin of the array, and not the address of the first word of data (called the ***data origin***). Our implementation never stores the data origin anywhere, and its value is computed using the virtual origin and the information in the array descriptor. The advantage of always using the virtual origin is that it makes array indexing operations fast. The disadvantage of not storing the data origin somewhere is that array copy operations require some recomputation.

One circumstance arises where arrays are represented slightly differently than the representation described above. When functions return an unconstrained array type, in addition to the descriptor and data (an object of type T_$full), the first word of the object returned is the virtual origin of the array. Composite function return objects are described below. Figure 4.6 illustrates the implementation of the procedure in Example 4.5.

4.6.4. Record Descriptors and Data

Records have the most interesting runtime type representation. Each record descriptor depends specifically on the declaration of the record itself. The BAC implementation recognizes several kinds of objects that may be record components. They are:

- A record discriminant, which is represented by a 4 byte integer in the record descriptor.
- A statically-sized record component, such as an integer, float, access type, or static array. These components are represented in the static portion of the record data, and since their location in the record data is known at compile time, they require no information in the record descriptor.
- A dynamically sized component that *does not* depend on the record discriminants. These include dynamic arrays, and constrained records that contain dynamic arrays. The data for such a component must be placed in the dynamic portion of the record data. Since the offset of this data varies between program instantiations, in addition to the data, the offset of the data in the dynamic part of the record data is kept in the record descriptor.
- A dynamically sized component that *does* depend on record discriminants. Again both records and arrays can fall into this category. Again the data for these components is placed in the dynamic part of the record data. In the descriptor, in addition to the offset of the data in the dynamic part, the descriptor of the

```
procedure array_example is
    type static_array is array(1 .. 10, 1 .. 20) of integer;
    a1 : static_array;
    n : integer := read_an_integer;
    type dynamic_array is array(1 .. n) of float;
    a2 : dynamic_array;
begin
    ...etc.
end
```

Example 4.5: Sample Procedure containing Array Types

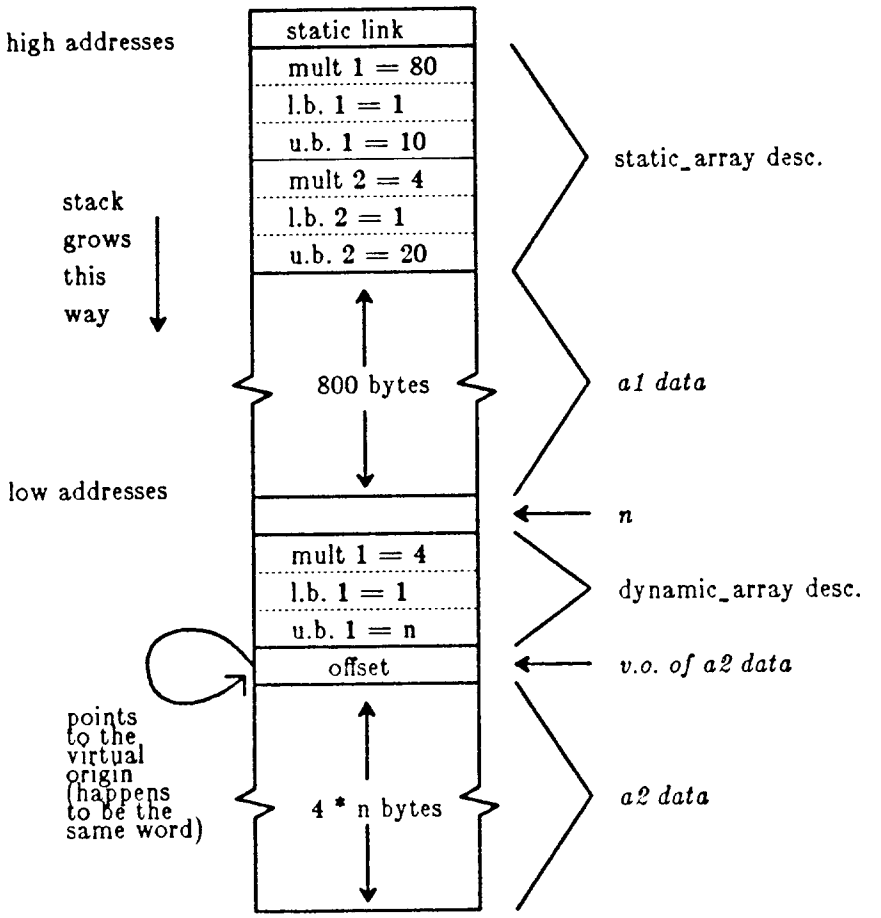

Figure 4.6: Structure of Stack Frame for Example 4.5

component is also stored.

- An unconstrained component. These can only be records. Unconstrained records are represented in records very much as they are in the stack frame, with an object of type T_$full allocated in the static part of the record data.

Example 4.6 will serve as a model to facilitate our discussion of record descriptors and data.

4.6.5. Structure of Record Descriptors

The structure of the record descriptor for Example 4.6 is presented in Figure 4.7. The absolute value of the first component of every record descriptor is the size (in bytes) of the record data. The size is used to avoid recomputation every time the data is copied. The sign of the first component tells whether a particular descriptor is the descriptor for a constrained object (sign is positive), or the descriptor for an unconstrained object (sign is negative). This information is used to implement the Ada 'CONSTRAINED attribute.

The next components of record descriptors are the record discriminants. All discriminants are represented in the descriptor as 4 byte quantities. All records that require descriptors have discriminants except a special case we call a ***subtype record***. A subtype record is an undiscriminated record that has components that are dynamic objects. Subtype records are a special case because while normal record types only need descriptors for constrained subtypes, subtype records have no discriminants and cannot be constrained, yet they require a descriptor. Thus, subtype records are the only ***types*** that are represented at runtime with a descriptor.

After all the discriminants in the descriptor, each record component that requires representation in the descriptor has space allocated for it. Static objects require no representation. Dynamic objects like string(1 .. m) and f(m) require that their offset into the dynamic part of the record data be stored in the descriptor. The descriptors for these objects remain constant during a particular subprogram activation, and are allocated and initialized separately. Objects that are constrained by the record discriminant (like string(1 .. rd) and f(rd)) require an offset because they are dynamically sized, and they also require a descriptor. This descriptor is provided because there is a potential need for the descriptor of any object, and the descriptor for discriminated objects can vary during a subprogram instantiation.

4.6.6. Structure of Record Data

The strategy of representing record data is close to the strategy of allocating data in the stack frame described earlier. The record data is divided into two contiguous parts: the static part, and the dynamic part. All static components are allocated in the static part of the record data, and all dynamically sized components are allocated in the dynamic part of the record data. The chief difference between data for records and data for a stack frame is that the offsets for the dynamic components of a stack frame are allocated in the static part of the frame, and the offsets for the dynamic components in the record data are allocated in the record

```
m : constant := read_a_constant;

subtype range1_to_3 is integer range 1 .. 3;

type f (fd : range1_to_3 := 2) is record
    f1 : string(1 .. fd);
end record;

type r (rd : range1_to_3 := 1) is record
    r1 : character;
    r2 : string(1 .. 10);
    r3 : string(1 .. m);
    r4 : f(m);
    r5 : string(1 .. rd);
    r6 : f(rd);
    r7 : f;
end record;
```

Example 4.6: Example Record Type with Various Component Objects

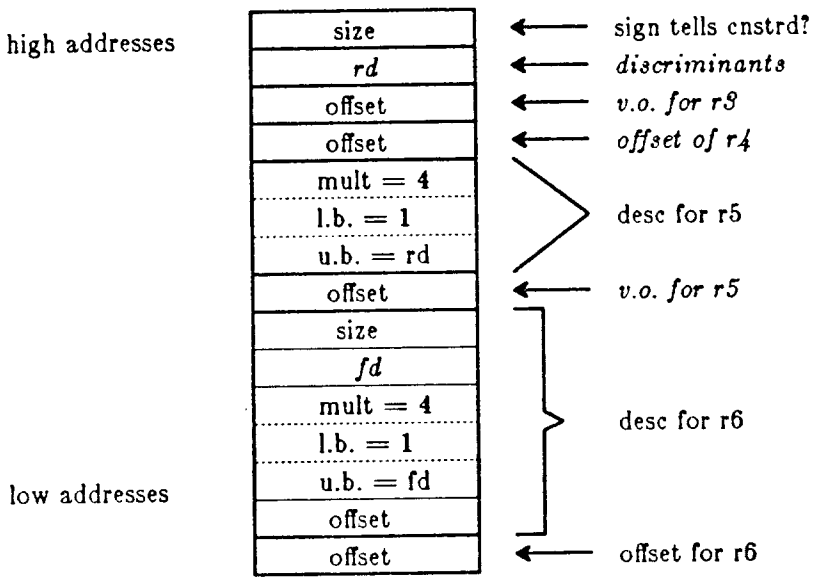

Figure 4.7: Structure of Record Descriptor for Type R in Example 4.6

descriptor, not the record data.

With this in mind, Figure 4.8 should be self explanatory. The only part of Figure 4.8 that may be puzzling is the allocation of space in the static part of the record data for unconstrained record object r7. This reflects the BAC approach to representing unconstrained records both as components and as frame objects. An unconstrained object of type T is represented by an object of type T_$full. A difficult problem arises with unconstrained records because they can be assigned from any object that has a legal value for its constraint. Implementations must handle the assignment of a larger object to a smaller one.

Various solutions to this problem have been proposed, including keeping unconstrained objects on the heap, and reallocating and copying if such an assignment

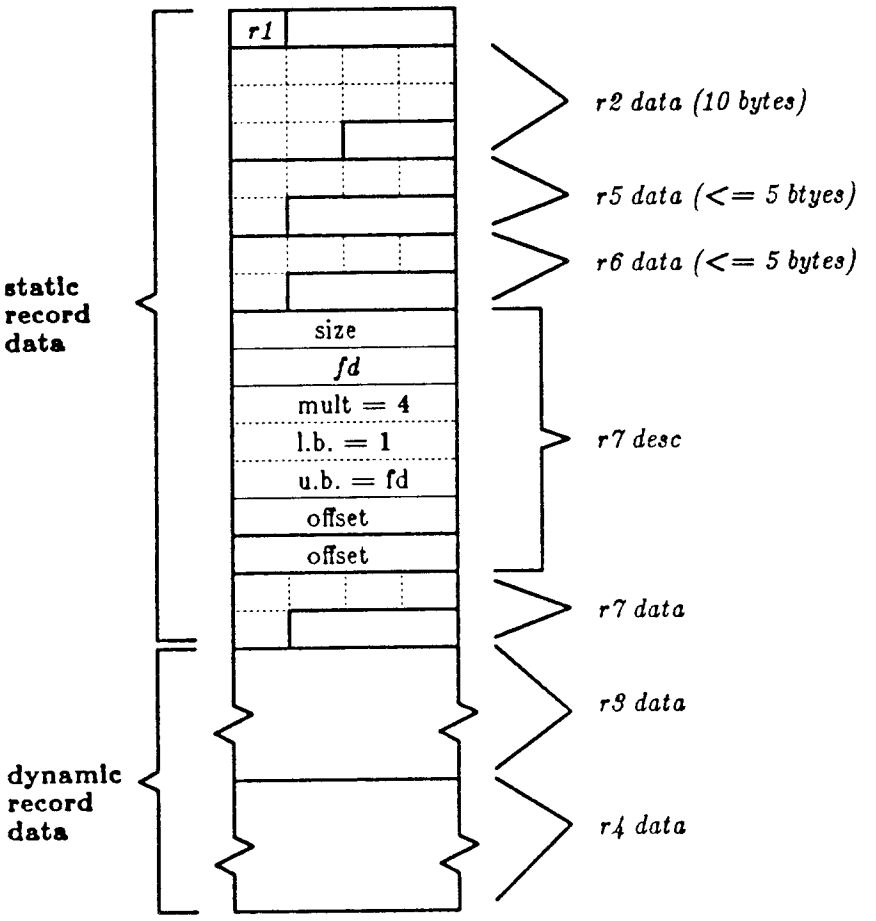

Figure 4.8: Structure of Record Data for Type R in Example 4.6

is made. The solution the BAC adopted is to allocate space for the largest possible discriminant value for each unconstrained record, and thus avoid having to reallocate and copy. Objects can never exceed this compiler determined maximum size and so can be allocated in the static part of the stack frame or record data.

This strategy has drawbacks. The Ada programmer must avoid unconstrained records that have potentially large discriminant values. In particular, the declaration in Example 4.7 would cause the BAC implementation to raise the STORAGE_ERROR exception. It is true that permitting programs like Example 4.7 will allow programmers to create true dynamic array variables[3], however, the performance disadvantages are high. We believe that if you give a user a feature, you expect him to use it. As a case in point, extensive use of dynamic strings would cause severe performance degradation, and we decided not to provide them. Furthermore, the Ada designers specifically avoided putting unconstrained arrays into Ada because they felt constrained array subtypes were sufficiently flexible that less efficient unconstrained arrays were unnecessary. This is one Ada implementation issue whose correct answer will become clearer after some experience.

4.6.7. Record Implementation Strategies

Because each record type will create descriptor and data objects with different structures, the problem of initializing and utilizing record information is non-trivial. The solution adopted in the BAC implementation is have the compiler create runtime subprograms associated with each record type definition. We call these compiler-provided functions ***thunks***. These subprograms are used to determine the size of a record object, initialize the descriptor of a record object, initialize a record object, and create an aggregate record object. Because certain record types are statically sized, or do not have descriptors, these thunks are not necessarily created.

```
type r (rd : integer := 5) is record
    x : string(1 .. rd);
end record;

bad_var : r;
```

Example 4.7: Record Type where the BAC Implementation Fails

[3] Array variables in Ada are required to be of a constrained subtype, and thus their size cannot vary during the instantiation of the subprogram in which they are declared. Array parameters *can* have an unconstrained type, but the formal parameter is constrained during each call any constraint on the actual parameter.

The thunks for record types take as their parameters the discriminants of the record type. In Ada, these discriminants can be used to constrain record component subtypes, or used as the tag field for a variant record. The BAC implements both of these uses, although the a variant record type requires defining a case statement in each thunk created.

Record initialization requires another kind of compiler provided function. Consider Example 4.8.

```
type r is record
    a, b, c : integer := f(x) + g(y);
end record;
```

Example 4.8: Record where Initialization Requires a JTHUNK

Because Ada semantics require re-evaluation of the initial value expression in a component list, in this case the expression f(x) + g(y), this expression must be packaged up in such a way that re-evaluation is efficient. The BAC approach to this problem is to create a runtime function, called a ***jthunk***, that can be called each time the expression needs to be evaluated. Because initial values are common, and because this technique is used to implement array and record aggregates, an efficient implementation of jthunks is important. The BAC implements jthunks without using the ***calls*** instruction, and instead has internal calling conventions that allow use of the VAX *jsb* instruction.

A more complete description of each of the runtime thunks is given in appendix A.

4.8.8. Reasons for Record Implementation Design Decisions

Several considerations affected the design of records. The first was the desire to make the runtime representation simple and uniform, which in light of Ada's parameterized records is not easy. We also sought to minimize the space overhead of the runtime record implementation, and avoid distributed overhead for undiscriminated records. Finally, we wanted to make runtime manipulation of records and descriptors as efficient as possible.

The implementation described above has the property that every record descriptor has a uniform structure. The greatest non-uniformity is the existence of subtype records, which were handled as a special case. With the amount of uniformity present, the thunks necessary to initialize record variables and allocate and initialize record aggregates were moderately easy to implement.

Minimizing space overhead was partially achieved by unifying all the descriptors for objects of a given constrained subtype. By keeping track and knowing the location of a particular descriptor at compile time, we reduced the overhead of associating a

descriptor pointer with each object. The original ABC implementation allocated a descriptor pointer with each object, and manipulated both descriptor and data every time any object was referenced.[Wet82]. The space and execution inefficiency of this implementation was a convincing argument against it.

Another way our implementation has the potential to save space is through dead variable elimination. We decided that we should layer the complexity of the code generation analysis and that the middle end should blindly assume all descriptors will be needed at runtime and allocate and initialize all of them. On the other hand, the middle end substitutes a constant for a reference to a descriptor if the descriptor's contents are known at compile time. Because of this substitution, many descriptors that are allocated are never referenced. An optimizing code generator would detect these dead descriptors and remove them. This removal implies that static arrays and non-subtype undiscriminated records will have no descriptors at runtime. This elimination achieves our goal of minimizing distributed overhead.

Our final goal was to improve execution speed. The implementation of Example 4.7 discussed above is one part of our solution. Another part is the use of offsets instead of absolute addresses to implement record descriptors. By using offsets, copies to undiscriminated records can be carried out as a block, instead of copying a field at a time.

4.7. Parameter Representation

Ada supports three kinds of parameters, **in**, **out**, and **in out**. The main difference between **out** and **in out** parameters is a semantic one, that **out** parameters cannot appear on the right-hand-side in any context. In this discussion, both **out** and **in out** parameters will be referred to as **out** parameters. Also, because access types in the BAC are represented with a 4 byte pointer, any discussion that applies to scalar types will apply to access types. The semantics of Ada allow either copy-result or pass-by-address parameter passing for **in** and **out** parameters. The BAC implementation adopts a hybrid approach, using copy-result parameter passing for scalar and floating point parameters, and using pass-by-address for composite object parameters.

4.7.1. Scalar and Real Parameters

In parameters for both scalar and real types are very similar. The value of the parameter is simply placed on the stack. Real parameters are somewhat unusual, simply because they are always widened (converted to double precision) before they are pushed.

Out parameters are implemented as a value-result mechanism, where both the address and the value of the parameter are pushed onto the stack, and the value location is used to store the intermediate parameter values as the called procedure executes. When the procedure returns, the stored address is used to update the value of the actual parameter. This action is compatible with Ada semantics, which require correct programs to assume that **in out** and **out** parameters are implemented in this way.

4.7.2. Composite Object Parameters

Array and record parameters are implemented by pushing the address of the object and referencing indirectly through it (pass-by-address). Unconstrained parameters are the most interesting composite object parameter. The semantics of Ada state that formal unconstrained parameters are constrained by the actual arguments at the point of the call. These semantics are implemented in the BAC by passing a descriptor argument as well as the address of the data object of the actual argument.

For **in** parameters, the descriptor argument is used for constraint checking inside the called subprogram. For **out** parameters, the descriptor can actually be assigned to, and is either used to make sure that no constraints were violated, or is copied back to the descriptor of the actual argument.

4.8. Function Return Value Representation

The BAC implements return values of scalar, access, and real functions in the traditional way. The result is placed in a register, or a register pair if it is a double precision floating point. Return values of composite objects are represented by allocating the object on the temporary stack and returning a pointer to it in a register.

The strategy of using an auxiliary temporary stack for function returns was adopted mainly for efficiency, because the nature of function values is mostly LIFO. The only tricky thing about function value returns is what happens when an exception is raised in a function that has been called and one of its arguments is a function value for which temporary space has been allocated. In this case, the temporary stack space may never be deallocated. The solution to this problem is to store with every exception handler a word that points to the place in the temporary stack that is guaranteed not to contain any usable function call results when the handler is entered.

A better strategy for statically sized function return values is to preallocate space for them on the runtime stack before the function is invoked. This preallocation avoids the necessity to deallocate them at a later time, and also avoids the exception raising problem discussed above. This strategy only applies to staticly sized objects, however, and the general strategy outlined above is necessary for dynamic objects.

4.9. Implementation of Other Ada Features

Ada has language features that require an implementation outside the type scheme that has already been described. These features include array slices, array aggregates, and record aggregates.

4.9.1. Implementation of Array Slices

Array slices in Ada are one-dimensional array objects that share the exact array data of the array being sliced. In the BAC slices are easily implemented by associating a new descriptor with the sliced array's data. The slice range values become the upper and lower bound of the descriptor. The BAC implementation allocates these

descriptors as compiler temporaries, and sets them up prior to executing code for the statement in which they occur.

4.9.2. Implementation of Aggregates

Aggregates present many problems to an Ada implementor because they can appear in different forms and contexts. Both record and array aggregates pose the re-evaluation problem described earlier in the discussion of record initialization. The expressions in them have a potential need for re-evaluation. This problem is solved in a similar manner, with the creation of jthunks for each aggregate value. The implementation of aggregates requires passing the addresses of these jthunks to an aggregate creation routine, which allocates space, sets up a descriptor, and initializes the aggregate.

4.9.3. Array Aggregates

Each dimension of an array aggregate can take on a distinct form, either positional, statically named, or dynamically named. These are illustrated in Example 4.9. The strategy of the BAC is to normalize these forms so that the function that initializes the aggregate only has to understand one form. This normalization requires considerable analysis.

The analysis is described for the one dimensional case, and the multi-dimensional case is a simple extension. We first describe the normal form. Each aggregate is initialized by a call to a runtime system routine. The parameters to this function include the address of the aggregate data, the address of the aggregate descriptor, and a set of tuples corresponding to the ranges of indices of the array where a particular value is to be placed. The format of each tuple is an upper-bound, lower-bound pair for each dimension, and the address of the jthunk to call to initialize all the elements

```
procedure aggregate is
    type foo is array(1 .. 5) of integer;
    m : integer := 5;
    x,y,z : foo;
begin
    x := (-1, -2, -3, -4, -5); -- positional aggregate
    y := (1 | 3 => 100, others => 102);
                          -- statically named aggregate
    z := (1 .. m => 500); -- dynamically named aggregate
end;
```

Example 4.9: Possible Forms of Array Aggregates

in the specified range.

The major job of the middle end is to take the different forms specified in the DIANA structure and normalize them. The positional aggregates are the most complicated. The tuples for positional aggregates depend on the context of the aggregate. In the simple case, each tuple would just indicate a single element of the array. For example, the tuples for the first aggregate in Example 4.9 would be (1,1), (2,2), (3,3), (4,4), and (5,5). The problem is that the lower bound of the tuples depends on the context of the aggregate. In some cases the type of the aggregate is not constrained which means the aggregate lower bound is not the lower bound of a constrained array type (as is foo, in Example 4.9), but instead the lower bound of an array base type (e.g. array (integer range <>) of integer). The point of this discussion is to indicate that a number of special cases exist when determining the bounds of positional aggregates. However, once the correct lower bound has been established, the tuples are just placed sequentially in the parameter list. The only other problem occurs when there is an **others** clause in the aggregate. To implement this feature, the compiler just generates a range from the current index to the aggregate upper bound.

Static named aggregates are similar to positional aggregates, except that they can contain ranges, and alternation (e.g. 1 | 4 => f(x)). The alternation is normalized into separate parts and sorted. The ranges (1 .. 4) are treated as range tuples (1,4). The only problem occurs when there is an **others** clause. The **others** clause fills in any array indexes that have not been accounted for. Fortunately, this is not difficult because the array index list must be known at compile time and is sorted.

The only dynamic expression allowed in an aggregate occurs when there is a single expression for an array dimension (e.g. 1 .. m => g(y)). Because they may only appear in this context, dynamic named expressions can be handled as a special case.

4.9.4. Record Aggregates

Record aggregates are created in a manner similar to array aggregates. The major difference is that because array descriptors and data are so regular, there is a generic array aggregate set up function, while with records, which differ widely among themselves, two different thunks are created with each record definition that correspond to setting up the record aggregate's descriptor and data respectively. Records aggregates have only named associations, and the use of **others** is more severly restricted, therefore the structure of record aggregates is easily normalized in the DIANA representation during the normalization phase.

CHAPTER 5

Conclusions

Conclusions will be presented in the following way. First, the experiences we had with DIANA and the C IR are summarized. Then the implications our runtime system design goals had on the actual BAC implementation are discussed. Next, performance measurements of the current BAC implementation are provided. With these figures we give reasons why performance data of the BAC and the ABC middle ends cannot be compared. Finally, the execution performance of Berkeley Pascal and the BAC is compared.

5.1. Conclusions about DIANA

Working with the DIANA representation was not a pleasant experience. The source reproducibility requirement of the DIANA design caused much of the DIANA structure to be unnormalized, hence more complex, larger, and less usable by the back end. In addition, DIANA is not particularly well designed for use by either the front end or the back end because important features like the symbol table are poorly represented. It is clear that the DIANA designers considered the environment tools the most important users of DIANA, and gave its space efficiency and compiler usability less consideration than they deserved.

5.2. Conclusions about the IR

Using the IR was a good decision. Because the IR provides a flexible low level representation that does not require the user to think about details such as register allocation, it is convenient and easy to use. Perhaps the greatest fault in the IR is that it is being used for more purposes than it was originally intended. The difference between the success of the IR and the failure of DIANA is clear. The IR is a form that was intended to ease retargeting C compilers to different architectures. It was not designed to be a low level representation for production quality compilers of many languages. The reason that the IR is so successful is that its value has been established through much experience with it. From its inception DIANA was designed to be a standard for Ada intermediate representations. However, this was decided long before experience with the DIANA representation had shown one way or the other that it is a good representation. The moral is that practice with any representation is the only way to determine its true value.

5.3. Conclusions about the BAC Runtime System

The runtime system described in this document was designed to be an efficient representation of the features necessary for a complete Ada runtime environment. Because the system shares as much descriptor information as possible, it does not

provide uniform access techniques for an object's descriptor. This non-uniformity means that the compiler has to be more intelligent about the context and type of a particular object. Thus, our runtime system attempts to achieve efficient representation at the cost of greater compiler complexity.

There are some problems with this approach. The increased complexity causes the middle end to be even larger than it already has to be to implement all the features of Ada. Our implementation of the middle end is approximately 20,500 lines of C source code, including the normalization phase and IR implementation. The Ada Breadboard Compiler middle end, which has C as its target language, contained approximately 6200 lines of C[1]. Their effort was not intended to be of production quality but the size difference is still significant. Since the middle end is the most difficult part of an Ada compiler to retarget, an important part of its design should be simplicity.

Still, there are clear advantages to our runtime representation. Sharing descriptors at runtime saves considerable stack space, and saves the execution time spent initializing redundant descriptors. Using an optimizer capable of dead-variable elimination, many of the descriptors that are present but not referenced will be eliminated altogether. With this representation, any Ada type declaration which would be legal in Pascal (i.e. static arrays, and non-discriminated records) would incur no runtime overhead not also present in the Pascal implementation. In this representation, none the overhead (type descriptors, thunks, jthunks, etc.) imposes a distributed execution overhead on programs that do not use the complex features.

5.4. The Performance of the Berkeley Ada Compiler

Some useful comparisons can be made between the Berkeley Ada Compiler and the Ada Breadboard Compiler. Due to the work of Murphy, the BAC DIANA representation is much smaller than the ABC's representation. Comparative statistics for a small test program[2] are provided in Table 5.1 and Table 5.2, which were adapted from [Mur84 pp.16-17]. Problems arise when one tries to compare the BAC and the ABC middle ends. Because we received an early version of their middle end, which we intended to reimplement, the state of the middle end we have for comparison is incomplete. In addition, efforts at AT&T have more recently gone into building a production compiler from scratch, so figures reflecting a complete version of their middle end are impossible.

Nevertheless, in an effort to make a meaningful analysis of the quality of the code generated by the BAC middle end, comparisons will be made with *pc*, the Berkeley Unix Pascal compiler. The BAC compiles source to object code at approximately 160 lines per minute, which is 2.3 times slower than *pc*. While part of this poor performance can be attributed to the complexity of DIANA, the middle end accounts for almost a quarter of the total compile time.

---

[1] This figure is somewhat suspect. See the discussion in the following section.

[2] The program was the ubiquitous Puzzle program of Forest Baskett, upon which seemingly thousands of analyses have unfortunately been based.

Puzzle Space Requirements		
246 lines of Ada source		
1635 input tokens (8 tokens/line)		
3218 DIANA nodes (16 nodes/line)		
1834 abstract syntax tree nodes		
921 symbol table nodes		
3073 sequences (list elements)		
	ABC	BAC
size of DIANA	254,816	140,276 bytes
average node size	72	35 bytes
external DIANA file	232	140 Kbytes

Table 5.1: Compilation Space Requirements for the Puzzle Program

Puzzle Compile Times on a VAX 11/750		
	ABC	BAC
front end	28.3	47.3 seconds
middle end	-	20.7 seconds
back end	-	24.8 seconds
total	-	92.8 seconds

Table 5.2: Compilation Execution for the Puzzle Program

Table 5.3 contains a comparison of execution times for 8 small benchmark programs. Perm generated all permutations of 7 objects. Towers solved the Towers of Hanoi. Queens solved the 8 queens problem. IntMM and MM did an integer and real matrix multiply, respectively. Puzzle has been introduced. Quick, Bubble, and Tree were all sorting algorithms. FFT solved a fast Fourier transform. Because the programs are so small, the significance of the comparison is questionable, but larger programs written in both Pascal and Ada are not available. The table shows the execution performance of the two compilers is comparable. The worst case for the BAC implementation was Perm, which suffered because it called a 3 line assembler function called Swap repeatedly. The overhead of passing the static link to the callee, and setting up the static linkage was the cause of the poor performance. A suggested modification of the simple static linkage model presented here would solve the problem, but also considerably complicate the model.

Program	Berkeley Pascal (sec)	Berkeley Ada (sec)
Perm	2.7	3.6
Towers	2.8	3.5
Queens	1.6	1.0
IntMM	2.2	2.0
MM	2.7	2.4
Puzzle	12.9	8.7
Quick	1.7	1.8
Bubble	3.0	2.0
Tree	6.4	4.6
FFT	4.8	4.3
Total	40.8	33.9

Table 5.3. Comparison of Pascal and Ada Execution Times

We view the favorable comparison with a language as relatively simple as Pascal as evidence that our runtime implementation was successful. In conclusion, we feel that the runtime system suggested in this paper provides efficient execution time performance with little distributed overhead, and offers a conceptually simple and useful runtime model for Ada.

Appendix A: Thunks needed in the BAC Implementation

This appendix contains a list of the compiler defined functions needed in the BAC runtime system. There are two classes of these functions, which I call thunks and jthunks. A thunk is specifically needed to implement parametrized records. It acts very much like a subroutine the user might have written, has its own local storage, and uses the *calls* instruction in its invocation. Callers of thunks provide their own lexical environment to the thunk so that it can access the necessary objects correctly. This implementation was chosen because these particular functions are called relatively infrequently. A jthunk is a function needed to implement specific Ada semantics which require the recomputation of expressions. A compelling example of the need for jthunks is given below.

```
procedure compel is
    x : array(1 .. 50000) of integer;
begin
    x := (1 .. 50000 => f(y));
end;
```

Example A.1: 50,000 Reasons for Efficient JTHUNK Implementation

Because Ada semantics require that f(y) be evaluated once for every element of the array x, efficient execution of f(y) is very important. The case where f(y) is replaced by a constant can be handled as a special case in any implementation, but the BAC implementation of the general case is so efficient that special casing constants is almost unnecessary.

Jthunks are implemented as pieces of code that the compiler branches to in order to evaluate a particular expression. The caller of the jthunk has to reserve a space in his stack frame for the address of the result of the jthunk. Thus the execution overhead of re-evaluating an expression is simply the cost of pushing the pc on the stack, and popping it off after the computation is over (*jsb* and *rsb* on the VAX), plus the extra cost of indirection in referencing the result. Because of jthunks are evaluated in the context of the caller they have the correct lexical environment.

THUNKS

All the thunks have as their first parameter the static link of the stack frame in which they are called. Also note that all of the thunks defined have to account for the case of a discriminated record whose discriminant is used as a variant. In this case, the thunk has to evaluate the equivalent of a case statement to determine which case is relevant.

rec_size
This thunk takes a parameterized record's discriminants, and returns the size of the data portion of the record. This particular information is placed in a field in the record discriminant so that later uses will not require recomputation. Undiscriminated, non-subtype records do not require *rec_size*.

rec_desc
rec_desc takes a parameterized record's discriminants, and the address of the base of the record's descriptor. The purpose of this thunk is to fill in the values of the descriptor given the discriminants. Consequently, *rec_desc* calls *rec_size*. Undiscriminated, non-subtype records do not require *rec_desc*.

rec_init
This thunk takes the address of the record data that need to be initialized. It is called to initialize the fields of a record that have initial values. Note that these functions will call possibly jthunks to revaluate arguments as in example 4.7, and will also potentially call other *rec_init* thunks. All records with initial values require *rec_init*.

rec_RDSU
RDSU stands for Record Descriptor Set-Up. This thunk initializes the descriptor for a record aggregate. It takes as parameters the address of the descriptor, and addresses of the jthunks that are evaluated to get the values of the discriminants. All records require *rec_RDSU*[1].

rec_RASU
RASU stands for Record Aggregate Set-Up. This thunk initializes the data for a record aggregate. It takes as parameters the address where the record aggregate data will be placed, the address of the record descriptor, and the addresses of jthunks that evaluate to the fields of the record. All records require *rec_RASU*.

JTHUNKS

aagg_expr
This jthunk is used to initialize array aggregates (aagg's) as in example A.1.

ragg_expr
This jthunk is used to initialize record aggregate (ragg's) components. The semantics of initialization of these is similar to array aggregates.

rinit_expr
This jthunk is used to initialize the components of a record that are declare with the same initial value, as in example 4.7.

---

[1] This is technically not true. If the compiler were to make a pass to determine whether any aggregates existed of each particular record type, then it could avoid generating these thunks for records for which there were no aggregates. This additional analysis seemed unnecessary.

References

[CoL82] R. P. Cook and I. Lee, "A Contextual Analysis of Pascal Programs", *Software—Practice & Experience 12* (1982), 195-203.

[DEC81] Digital Equipment Corporation, *VAX-11/780 Architecture Handbook*, Digital Equipment Corporation, 1981.

[Cor81] J. R. Cortopassi, "RX, A RIGEL Interpreter", Master's Thesis, Computer Science Division, EECS, UCB, Berkeley, CA, 1981.

[DoD83] U. S. Department of Defense, *Military Standard — Ada Programming Language*, U. S. Government Printing Office, Washington, DC, 1983. ANSI/MIL-STD-1815A-1983.

[GWE83] G. Goos and Wm. A. Wulf, eds.; Arthur Evans, Jr. and Kenneth J. Butler, rev., eds., *Diana Reference Manual, Revision 3*, Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[Hen84] R. R. Henry, "Graham-Glanville Code Generators", PhD Dissertation, UCB/CSD 84/184, Computer Science Division, EECS, UCB, Berkeley, CA, May 1984.

[Ich79] J. D. Ichbiah, "Preliminary Ada Reference Manual", *SIGPLAN Notices 14*, 6 (June 1979).

[Hon80] Honeywell, Inc., *Formal Definition of the Ada Programming Language*, Cii Honeywell Bull, INRIA, 1980.

[Kes83] P. B. Kessler, *The Intermediate Representation of the Portable C Compiler as used by the Berkeley Pascal Compiler*, Computer Science Division, EECS, UCB, Berkeley, CA, April 1983. Unpublished Manuscript.

[Lam83] D. A. Lamb, "Sharing Intermediate Representations: The Interface Description Language", PhD Dissertation, CMU-CS-83-129, Computer Science Dpt., CMU, Pittsburgh, PA, May 1983.

[Mur84] M. P. Murphy, "DIDI - Diana Implementation Designs and Decisions", Master's Thesis, Computer Science Division, EECS, UCB, Berkeley, CA, 1984.

[Qui82] M. E. Quinn, *The Ada Breadboard Compiler: The DIANA Package*, Bell Laboratories, Murray Hill, NJ, 1982. Internal memorandum.

[QuW83] M. E. Quinn and C. S. Wetherell, *An Intermediate Representation for the BTL Ada Compiler*, Bell Laboratories, Murray Hill, NJ, 1983. Internal memorandum.

[Rub82] D. H. Rubine, *A Hybrid Ada Interpreter*, Bell Laboratories, Murray Hill, NJ, 1982. Internal memorandum.

[Wet82] C. S. Wetherell, *The Ada Breadboard Compiler: An Overview*, Bell Laboratories, Murray Hill, NJ, 1982. Internal memorandum.