A Gypsy-to-Ada Program Compiler

Robert L. Akers

December 1983 Technical Report 39

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

Abstract

Gypsy is a language for formally specifying, implementing, and proving computer software systems. The language has been carefully designed so that it is always possible to construct logical formulas which are sufficient to show that a program satisfies its specifications whenever it runs. Ada is a programming language that has been designed for the implementation of embedded systems. In the future, Ada will be the required implementation language of many critical software systems. Formal specification and verification of these critical systems is highly desirable, but there are substantial parts of Ada that are not susceptible to practical specification and proof methods.

One solution to this problem is to use Ada as a target language for a Gypsy compiler. Critical software systems can be formally specified, implemented, and proved in Gypsy using the full set of tools available in the Gypsy Verification Environment. Then the proved software system can be compiled into Ada. This approach has several important advantages. First, it makes maximal use of the existing Gypsy verification tools. Second, it allows Gypsy programs to run on any machine that runs Ada. Third, and most important, it effectively decouples the development of Gypsy and Ada.

The subject of this report is the design and initial implementation of a Gypsy compiler that uses Ada as its target language. This compiler is available as one of the tools in the Gypsy Verification Environment.

Acknowledgements

Don Good, Rich Cohen, Larry Smith, and Mike Smith assisted in the selection of the compiler as a thesis topic. Don Good was primarily responsible for acquiring funding for the project.

Bill Young assisted in the implementation of dynamic structured types by coding the Ada support packages of standard functions for those types. Discussions with Bill Young and Don Good were invaluable in arriving at the final design for the implementation of dynamic types. John McHugh made significant contributions in discussions on the implementation of condition handling.

Larry Smith was my constant sounding board. His contribution cannot be underestimated, nor my appreciation sufficiently expressed.

All of these people have made valuable suggestions and provided numerous insights in discussions which helped formulate the implementation.

Chapter 1
INTRODUCTION

Gypsy [11] is a language developed to support a methodology for formally specifying, implementing, and verifying computer software. Developed from Pascal [18,35], it is a concise, clean, and very modular language in the lineage rooted in ALGOL 60 [23]. The evolution of Gypsy from Pascal was largely a process of eliminating those language constructs which did not lend themselves to verifiability, replacing them with more tractable alternatives, and adding some new capabilities, including a mechanism for the formal specification of program behavior.

The Gypsy Verification Environment [6,7,8] is an interactive system of tools, implemented in LISP, which work together to aid the programmer in executing the tasks of incremental program development and verification. Programs may be incrementally specified, implemented, parsed, proven, and compiled, all within this environment.

Ada [16,32] is a programming language of considerable expressive power designed to be applicable to embedded systems and a wide domain of other applications. Ada also has its roots in the heritage of ALGOL and Pascal, and bears many syntactic and semantic resemblances to Gypsy, though Ada is much larger and more complex. The critical nature of some of its applications has aroused interest in the possibility of verified Ada programs, but certain aspects of the language preclude of formal verification [31,37].

The subject of this report is the design and initial implementation of a Gypsy compiler which targets to Ada. The compiler is the combination of a LISP module integrated into the Gypsy Verification Environment and packages of support routines written in Ada. The compiler converts an internal form of a Gypsy program maintained by the verification system into Ada source, which may then be compiled along with the support packages in any Ada environment. Using the Gypsy verification methodology and tools to perform the actual task of program verification and then applying the compiler to obtain Ada code with the same semantics results in Ada code which may be used with much the same confidence as the verified Gypsy program.

There are several aspects of this project, some pragmatic and some theoretical, which are attractive to those interested in formal program verification. First, this approach will require minimal time and expense in producing verified Ada, since the tools it utilizes for program development and verification already exist and have demonstrated success. Such a system of tools is normally huge and costly, and producing a whole system tailored to Ada might be prohibitively expensive. Secondly, this approach will provide a mechanism for getting Gypsy programs to run on any machine which can run Ada. Thirdly, this approach effectively decouples the development of Gypsy and Ada. To users, this means that Gypsy is a verifiable systems development language which can compile into Ada. To the verification research community, this means verification research may be cleanly decoupled from Ada. The importance of this decoupling is that it allows subsequent development of Gypsy and Ada to proceed without mutual interference.

This report will briefly outline background to the compiler effort, describe the mapping from Gypsy to Ada constructs, and describe the role of the compiler in the Gypsy Verification Environment and the high points of its implementation.

1.1 Background -- Compiling to High Level Languages

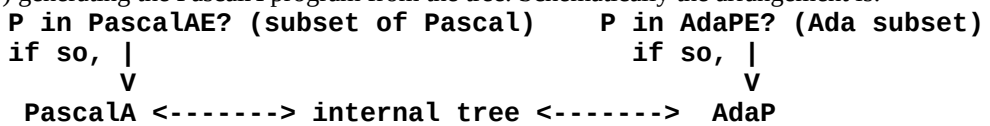
Usually, programs are compiled from a high level language to some machine language, but there are often reasons for compiling programs from one high level language into another. For example, an enterprise may wish to consolidate its software effort to a minimum number of languages so as to streamline and simplify its personnel assignments. Two other cases, the ones motivating this report, are: 1) that special program development strategies available for one language do not exist for the language in which the program must ultimately be compiled, and 2) that a program written in some language may be needed on another machine for which no compiler for the language exists.

Source-to-source compiling from one high level language to another is not nearly as common as compiling to a lower level language, nor has it received as much attention in the literature. Nevertheless, it is not without precedent. One well documented effort, which is germane to this report, is the Pascal-to-Ada and Ada-to-Pascal translator implemented by the PascAda group at the University of California at Berkeley [2]. The problems of translating from Pascal to Ada are quite similar to those of translating Gypsy to Ada. (The project documented in this report is not concerned with problems analogous to compressing features of the Ada language into the less baroque Pascal. The questions of reflexive translation, whereby a program translated from Gypsy to Ada and back to Gypsy needs to resemble its original form, is never raised here.)

The central strategy of the PascAda group was to define compatible subsets of Pascal and Ada which have a direct translation between them. A program written in full Pascal is transformed into a form using only those constructs in the Pascal subset. The program in Pascal subset form is then converted into the Ada subset form and regenerated as Ada source for interpretation or compilation. The converse process is followed for translation from Ada to Pascal. The target, then, is the subset language rather than the full version of the target language. Where AdaP is the Ada subset and PascalA the Pascal subset, the general schema of the translation is:

While PascalA and AdaP are semantically equal, primarily syntactic issues will make the incarnations of a single program look somewhat different in the two languages. To aid in the transition, a single intermediate form was selected to represent the semantics of a program in either language. This form could be used as a basis for generating the syntactically appropriate source code in either language. Translations to and from the intermediate form are generally local in nature. The intermediate form chosen was the tree structure defined in the formal definition of Ada [9].

Some features of the each language have no analog at all in the other language, and these were thereby excluded from translatability. (This is primarily true for Ada features which do not map into the simpler Pascal.) A translatable subset, larger than the semantically equal subsets, was defined for each language. These were called AdaPE for the Ada side and PascalAE for Pascal. The process of translating a program P, say from Ada to Pascal, then became: 1) determining if P is in AdaPE (is translatable), 2) transforming P (an AdaPE program) into AdaP, 3) translating the AdaP representation of the program into the intermediate tree structure, and 4) generating the PascalA program from the tree. Schematically the arrangement is:



Since the intermediate tree form was designed to represent Ada programs, the algorithm to prettyprint the Ada source is a straightforward, top-down, post order traversal, which prints text as it goes. (This is quite similar to the operation of infprint on Gypsy prefix representations in the Gypsy Verification Environment.) Since the tree representation was designed for Ada, some information often needed to be collected or inferred in order to print Pascal from the tree. (Similar collection is necessary in generating Ada from the modified prefix which serves as the translation medium for the Gypsy-to-Ada compiler.)

1.2 Goals And Priorities

The implementation of the Gypsy to Ada compiler is an ongoing project. The work documented here is the attempt to compile the bulk of the Gypsy language, with the notable exception of its mechanisms for concurrency. It may be reasonably expected that work on concurrency may follow, but our goal has been to compile accurately the large subset of Gypsy which does not involve concurrency before expanding to the full language.

We realized that on occasion there would be certain uses of Gypsy language features which would not lend themselves at all to the compilation effort, or for which the overhead of compilation would exceed the benefits. The compilation philosophy used in this report allows for the acceptability of variances where such cases might arise.

The primary role of the compiler is simply to aid in producing running object code from Gypsy programs, which may have been completely verified. It need consider, then, only those parts of the program which will form its runtime image. All code which contributes only to the verification process or otherwise to program development may be disregarded. To the greatest extent feasible, remnants of untranslated Gypsy text, for example non-validated formal specifications and lemmas, are maintained as comments in the Ada code.

Our approach concerns only the one-way compilation from Gypsy to Ada. The task of compiling Ada programs into Gypsy is much more difficult, due to the generality of Ada. Some of Ada's fundamental characteristics which do not lend themselves to verification might likewise not lend themselves to representation in a language which is restricted by the rigors of verifiability.

It is hoped that the results of this effort will serve as fruitful ground for the further study of language-related issues regarding Gypsy and its usage, as well as its relation to Ada.

Clearly, the foremost priority of the compiler is that the Ada program must be functionally equivalent to the Gypsy program so as to preserve the validity of the Gypsy proofs. A secondary priority is to maintain a physical resemblance of the Ada program to the original Gypsy. It is understood that this resemblance must dissolve in the compilation of many constructs which are treated differently in Ada, even though the differences may be subtle. A less significant priority is the efficiency of the Ada program.

1.3 Caveat On Ada Source Modifications

Since the validity of the verification of a Gypsy program is extremely sensitive to changes in the program, the Gypsy Verification Environment (GVE) includes a facility for tracking the effects of such changes through incremental program development. This facility notes which verification conditions will have to be regenerated after a change and which unchanged ones will have to be re-proven in order to maintain the validity of the verification. Changes occurring outside the GVE impose a complete caveat emptor on the verification status of the program.

A verified program may undergo several translations, all of which are designed to preserve verification status. These translations, supervised in the GVE, may include compilation via translation to BLISS for later BLISS compilation, or the translation to Ada described here. The role of any Gypsy compiler is to produce a functionally equivalent program in its target language. A program emitted from a compiler may be considered to be "verified" only if the Gypsy program being translated was completely verified in the GVE and was unchanged since the verification was performed. Similarly, the code produced by a compiler, be it BLISS or Ada, must not be altered. To do so would invalidate the verification of the program.

Chapter 2

THE ADA IMAGE OF GYPSY CONSTRUCTS

This chapter describes the Ada forms into which the various Gypsy constructs are compiled. It parallels the text of the Revised Report on the Language Gypsy Version 2.1, by Donald I. Good, [38] which is the current defining document for Gypsy. The treatment of Chapters 1-7 is the most thorough, since the compiler was designed first to completely translate the features described in those chapters. Material from the other chapters is covered to the extent to which the compiler software currently treats it.

2.1 Basic Concepts

The broad concept of a program in the two languages is identical, although Gypsy buffer parameters to the main procedure are treated directly as files in the Ada environment. The fundamental concept of procedure, function, type, and constant are similar in Gypsy and Ada. Gypsy lemmas, since they have no runtime significance, may be transformed into comments. Dynamic memory is implemented with explicit pointers, although this implementation is totally abstracted from user control in Gypsy.

The Gypsy standard types boolean, integer, and character are present in Ada, and the set of predefined operations for objects of those types are similar, but not identical. Where there are differences, Ada support code has been written to perform exactly as would the Gypsy program. Rational numbers, as currently implemented in Gypsy, serve no runtime purpose, so they are not translated. The type activationid is not relevant to the Ada image as currently implemented. Scalar types are supported as enumeration types in Ada, and additional support provided in the compiler ensures that their implementation reflects Gypsy semantics. Ada includes the standard composition types array and record. Ada's access types are used in conjunction with records to implement Gypsy sets, sequences, mappings, and buffers. Strings are present in Ada, and string constants in Gypsy and Ada share the same notation. Ada's predefined strings are implemented as arrays of character, however, while Gypsy's are sequences of character. The compiler implementation must treat them as it does sequences, which includes coercing constants from array to sequence (linked record) value.

Gypsy's standard procedures are emulated in three ways: 1) as routines in instantiations of predefined generic support packages, 2) as routines in a non-generic support package, or 3) with in-line expansions possibly incorporating calls to the above types of support routines. Gypsy's IF, CASE, and LOOP statements have counterparts in Ada with which the Gypsy semantics may be expressed.

The implementation preludes (see Ch. 2.5.9.3) which will be available for use with the Ada compiler will mainly provide support for the input/output mechanisms. They will include type and object representations which will represent internal objects eligible for treatment as input/output devices. When the user invokes the compiler from the GVE monitor, he will supply a Gypsy-like call to the user's main procedure which will link the program to an environment and drive the compiler's treatment of input/output.

2.2 Lexical Preliminaries

The character set available to Ada programs is the same as that for Gypsy. Ada identifiers have the same form as Gypsy identifiers. The bound on identifier length is an Ada implementation constraint. The set of reserved words is somewhat different in the two languages. Where a Gypsy identifier is the same as an Ada reserved word, it is transformed through translation to some non-reserved identifier. The set of identifiers beginning with the characters "g_a_" are reserved for use by the compiler.

Any construct translated as a comment becomes an Ada end-of-line comment, textually preceded by "--" on every line. A <CRLF> is inserted at the end of the comment.

2.3 Type Specifications

The concept of type is quite similar in Ada and Gypsy. The syntactic form of type specifications in Ada is quite similar to Gypsy's, and all varieties of Ada types are present in Gypsy with the exception of the access type. The fundamental target mechanisms for translating Gypsy types, then, are present in Ada. Even so, the translation of types and their predefined operations and the provisions made for replicating Gypsy type compatibility in Ada is one of the most involved processes undertaken in the compiler.

While both Ada and Gypsy are strongly typed languages, Ada places constraints on type compatibility which are not present in Gypsy. Type compatibility in Gypsy means only that the actual parameters in function and procedure calls must be of the same basetype as the formal parameters. (The same treatment is given to arguments to the various predefined statements.) Type compatibility is all that is required by the Gypsy parser for type checking. Checks on range and size restriction violations on the types of formals must commonly be deferred until runtime, when a violation will result in the signalling of the VALUEERROR condition.

In Ada, however, type compatibility means that the objects are either of the same declared type or are of explicitly declared subtypes which share the same basetype. For example, consider:

```
type int1 is integer;
type int2 is integer;
type int3 is integer range 1 .. 10;
```

Objects of any pair of these types are not type compatible with one another, since each type declaration represents a distinct data type. In order for them to be compatible, each must be declared as a subtype of type integer. Consider:

```
subtype int1 is integer;
subtype int3 is integer range 1 .. 10;
```

Objects of types int1 and int3 are type compatible both with each other and with objects declared to be of predefined type integer.

To achieve the Gypsy sense of type compatibility in the Ada program, then, it is necessary to declare explicitly the basetype of every type declared in the Gypsy program, and to declare each user-declared type to be a subtype of the appropriate basetype. Moreover, any objects sharing the same Gypsy basetype need to be linked to the same Ada basetype. Each basetype is declared only once in the Ada program. Consider the Gypsy type declarations in scope foo:

```
type array1 = array ([1..10]) of integer[1..10];
type array2 = array ([1..10]) of integer[0..1000];
```

These two array types will share the same basetype, which will be declared in BASETYPE_PACKAGE to be:

```
type array1_foo is array (integer range 1..10) of integer;
```

Both array1 and array2 are declared to be subtypes of array1_foo.

A further rationale for the use of this basetype mechanism relates to predefined operations in Gypsy. These

include predefined operators, standard functions, and predefined statement forms such as assignment. In many cases Gypsy provides operations which Ada does not, and in many other cases the operations provided have different semantics than their predefined Ada counterparts. The compiler must provide Ada support semantically equivalent to that of Gypsy. Strong typing in Ada suggests that these operations would be most efficiently defined for the various basetypes which occur in the program. (The Ada generic capability is not general enough to provide, for example, a single assignment procedure for all objects, but it may be used to provide a single assignment procedure for all integer types.) Any operation properly defined on the basetype of its operands may be used with operands of any subtype, provided a mechanism exists for providing appropriate behavior regarding subtype restrictions.

This mechanism involves the use of a typedescriptor. Each Gypsy type has associated with it a typedescriptor constant which describes it in terms of its kind (i.e. sequence, set, array, etc.), its range and size restrictions, and typedescriptors of its component types. A typedescriptor will be provided as a parameter to all functions and procedures which must be sensitive to range and size restrictions, for example where runtime valueerror or indexerror checking is performed. The typedescriptor type is declared in the Ada program to be a variant record, with the kind of Gypsy object as its discriminant. For an example of a typedescriptor, consider these Gypsy type declarations:

```
type smallpos = integer [1..100];
type small_array = array ([1..10]) of smallpos;
```

The typedescriptor declarations which would accompany the translations of these types are:

```
smallpos_typedescriptor: constant gypsy_package.typedescriptor
      := (kind => g_integer; low => 1; high => 10);

small_array_typedescriptor: constant
      gypsy_package.typedescriptor := (kind => g_array;
          index_type => (kind => g_integer; low => 1;
          high => 10); elem_type => smallpos_typedescriptor);
```

For a declaration of the typedescriptor type, see Appendix A.

The predefined operations on basetypes are provided in four different ways, depending on which is most appropriate. If the correct operation exists in the predefined Ada support, for example with integer comparisons, it may be used in its native form. The remaining three alternatives are generated during translation. For basetypes which are predefined in the compiler, i.e. integer, character, boolean, sequence of integer, sequence of character, and sequence of boolean, the predefined operations are provided in a support package named GYPSY_PACKAGE. For those other basetypes which result from user declared types, the functions are provided either through instantiation of generic packages of routines germane to a kind of type (array, sequence, set, mapping, buffer), or, where the routines do not lend themselves to generic instantiation, by in-line declaration based on templates embedded in the compiler. This latter method is used most notably on enumerated scalar and record types, which are n-ary in nature, and on functions supporting the seqconstructor and setconstructor operations. The listings for GYPSY_PACKAGE and the generic basetype packages are provided in Appendix A and Appendix B.

A feature of Ada very unfortunate for this project is that any program unit or object must be declared before it may be referenced. Since Gypsy scopes are translated into Ada packages and Ada packages must include in their headers the names of all other packages which they reference, there is a constraint on Gypsy input disallowing scopes which form a ring of inter-scope references via name declarations. This can create difficulties for programs not composed with this constraint in mind. Declaring a basetype in the same package as one of its subtypes, and then providing access to that package for all other packages using any other subtype of the basetype could greatly magnify the difficulty.

To simplify this problem, the compiler places all basetype declarations in a separate package, BASETYPE_PACKAGE, which appears first in the translated code. All other packages are declared with access to the basetype package. The basetype package also includes a valueerror checking function, default initial value and typedescriptor constant declarations, and the declarations of all the standard functions provided by Gypsy on objects of each basetype. Grouping all basetype declarations and their associated unit declarations

in an isolated part of the translated program has the desirable side effect of keeping directly translated code free of constructs radically different in appearance from the original Gypsy program.

For an example of a basetype package resulting from translation of a sample Gypsy program, see Appendix D.3.

2.3.1 Initial Values

Types in Gypsy have default initial values associated with them, while types in Ada do not. To provide an initial value, each type declared in translated code is accompanied by a constant declaration which specifies the initial value for the type. If the user has provided an initial value in his Gypsy type specifications, then that value is the one given as the value of the constant. Otherwise, the Gypsy default initial value for the type is used. Composed types have initial values based on the default initial values of their component types. Array and record initial values may, but need not, be expressed as aggregates. Dynamic types must have their initial values expressed in terms of calls to predefined functions which construct objects of the type. The default initial value for a dynamic object is a record structure with null item pointers.

For example, consider the following type declarations.

```
type foo_int1 = integer [1..10];
type foo_int2 = integer [1..100] := 100;
type foo_array = array ([1..10]) of foo_int1;
type foo_seq = sequence of integer;
```

The corresponding initial value constant declarations would be:

```
g_a_foo_int1_initial: constant foo_int1 := 1;
g_a_foo_int2_initial: constant foo_int2 := 100;
g_a_foo_array_initial: constant foo_array := (1..10 => 1);
g_a_foo_seq_initial: constant foo_seq :=
      basetype_package.foo_seq_basetype.null_value;
```

These constant declarations appear with the type declarations and may be used in any package which may refer to the type name.

All Gypsy variables are assigned initial values either explicitly in declarations or implicitly from the default initial value of the type. Their Ada counterparts must be given initial values explicitly. Consider the following declarations.

```
foo1: foo_int1;
foo2: foo_int2 := 10;
foo3: integer;
```

These will be translated to:

```
foo1: foo_int1 := g_a_foo_int1_initial;
foo2: foo_int2 := 10;
foo3: integer := 0;
```

2.3.2 Simple Types

The Gypsy simple types boolean, character, integer, enumerated scalar types, and subrange types are all provided in Ada, but some operations on objects of these types must be treated differently, and thus generated by the compiler. The operations eq, ne, lt, le, gt, ge, max, min must be defined for all simple types. The functions lower and upper must be defined for all bounded simple types. An assignment procedure must also be provided which responds to bounds violations by raising a valueerror rather than the Ada constraint_error.

Eq, ne, lt, le, gt, and ge are all defined in Ada and may be used in their infix form. The functions max and min are defined in GYPSY_PACKAGE as generic functions operating on range operands. Where T is a

typename, lower(T) and upper(T) may be transformed into Ada's functional attributes T'first and T'last. Assignment is generated in a manner appropriate for each kind of type.

2.3.3 Scalar Types

The basetype of a scalar type is the scalar type itself. Since there may be any number of items in the value set, the type declaration may not be parameterized for generic treatment. Therefore, scalar type declarations are generated from templates and appear in BASETYPE_PACKAGE. The operations pred, succ, ord, and scale must be defined, and they, too, are generated from templates, along with the assignment procedure and valueerror checking function.

The typedescriptor for an enumerated scalar type is given in terms of the ordinal position of the lower and upper bounds of the type within the sequence of scalar identifiers composing the type.

2.3.3.1 Type Boolean

Gypsy operations defined for type boolean are: not, and, or, imp, and iff. Of these, not, and, and or are sufficiently expressed in Ada. Since type boolean is one of the predefined basetypes known to the compiler, imp, iff, assignment, and the valueerror function are defined in GYPSY_PACKAGE. The universal and existential quantification operations are not intended for execution, so they are not translated. If they are detected in executable code, the translation fails. In non-executable code, they are printed in their original form for documentation purposes.

2.3.3.2 Type Character

The type character is predefined in Ada and carries the same operations as scalar types. Since character is one of the predefined basetypes known to the compiler, the pred, succ, ord, and scale functions and the assignment procedure are declared in GYPSY_PACKAGE. Character literals are printed in the form consistent with the Ada declarations of the CHARACTER and ASCII packages. Printable characters are printed using the quoted form; control characters are printed using the constant identifiers from the ASCII package.

2.3.3.3 Type Integer

The Gypsy operations on integers include "+", "-", "*", "**", div, mod, and unary "-". Ada supports all of these operations (Ada "rem", not "mod", is the Gypsy "mod".), but the only exceptions which can be raised by the Ada operators are CONSTRAINT_ERROR and NUMERIC_ERROR. Contrast this with the more specific conditions raised by the Gypsy operators, which include adderror, subtracterror, multiplyerror, powererror, powerindeterminate, negativeexponent, divideerror, zerodivide, and minuserror. So that the proper exceptions will be raised by the integer operations, the compiler provides each of the operations as a predefined function. Typically, these functions will have the call to the appropriate Ada operation embedded and will trap Ada exceptions with the appropriate granularity, converting them to the various Gypsy conditions. The functions, along with other declarations germane to integer support, are defined in GYPSY_PACKAGE. Note that they are declared as functions and not as overloaded operators. This means the infix operator forms will be translated into function calls with normal syntax.

Gypsy 2.1, now under development, will include notation for representing integer literals in a binary or octal form. The Ada compiler will convert these numbers into decimal form.

2.3.3.4 Type Rational

The Gypsy verification system does not support rational numbers in executable code. They exist in Gypsy as a specification tool. If the compiler discovers a rational number in executable code, it will issue a message and disallow the translation. Note that executable code may include runtime validated specifications, for which executable code must be generated to validate the runtime program state. If rationals appear in non-executable specifications, they appear as normal in the commented specifications.

2.3.3.5 Subrange Types

Subrange types are fully supported in Ada with only minor variation in notation. As previously stated, to duplicate Gypsy type compatibility notions, subrange types must be declared as subtypes. The idea of pre-computability is not discussed in standard Ada documents, but it is reasonable to assume that the Gypsy notion of pre-computability, narrow as it is, could be supported by an Ada implementation.

As long as subrange types are declared with type declarations, they are completely consistent with the initial value and typedescriptor mechanisms utilized for other types in the compiler. A problem arises, however, when symbols are declared to be of a type with the subrange attached to the symbol declaration rather than a type declaration. For example, consider:

```
var foo1: integer [1..10];
{ This is opposed to:
      var foo2: smallpos;
  where type smallpos = integer [1..10]; }
```

To determine the typedescriptor of foo2, the compiler needs only recognize that foo2 is of type smallpos and then look up the name of the typedescriptor for smallpos. Foo1, however, does not have an independently declared type, hence there is no declared typedescriptor for foo1. To remedy this situation, the typedescriptor is generated as an aggregate expression whenever it is needed, for example in an assignment to foo1. The GVE may soon be giving these objects of anonymous subrange type special treatment by declaring invisible types. When this treatment is provided, the generation of these aggregates on demand will no longer be necessary.

2.3.4 Static Structured Types

Ada has array and record mechanisms which are sufficient as targets for translating Gypsy arrays and records, provided some predefined operations are superseded by new definitions.

2.3.4.1 Arrays

Gypsy arrays map neatly into Ada arrays with a minimum of overhead. The support for an array basetype is entirely declared in a generic package which may be instantiated for each basetype. The units declared in the package include the basetype typedescriptor, the basetype itself, the element selector function, the element alteration function, a valueerror checking function, an aggregate assignment procedure, and an element assignment procedure. The element selector, aggregate assignment, and element assignment procedure are different from those in Ada because of their distinct handling of exceptions. The element alteration clause, as with all alteration clauses, must be provided because no analogous operation is predefined in Ada. Equality and inequality are adequately handled in Ada.

Having these various operations declared as functions and procedures produces quite a notational impact on translated code which utilizes them. Handy and easily recognizable forms such as [i] for indexing and ":=" for assignment must be abandoned in favor of standard function and procedure calling forms. Even these are somewhat unsightly due to the use of qualified global names which are used throughout the compiler. For example, the simple reference to a[1], where a is of type foo, becomes:

```
basetype_package.foo_basetype.select_element
                              (a, foo_descriptor, i)
```

The arguments to the select_element function are the array, its typedescriptor, and the index expression. The global name of the select_element function must include the name basetype_package, from which it was generically declared, the name of the array basetype (here assumed to be foo_basetype), which is the name of the instantiated array basetype package, and the name of the selector function in the package. This is not so much new complexity as it is hidden complexity brought to the surface through translation. Nevertheless, the unfortunate truth is that code which deals heavily with highly structured data types becomes very difficult to read, particularly if the reader is not well-versed in the compiler's operation.

2.3.4.2 Records

The operations defined on Gypsy records are eq, ne, field selection, assignment (both of aggregates and of individual field values), and field alteration. Eq, ne, and field selection are sufficiently handled in Ada. Both varieties of assignment and, of course, alteration, must be provided by the compiler for each record basetype. This involves much more overhead than immediately appears.

Records are the bane of generics in Ada, largely because record field names cannot be used as expressions for parameterization and also because there may be any number of fields in a record. Generic packages could not be useful in generating support code for records, so those functions are generated from templates embedded in the LISP code of the compiler.

Not only may generics not be used, but a function which must deal with an individual record field must be written to operate specifically on that field. Two cases where this complicates the translation are in generation of field assignment code and in generation of alteration clause code. Each field in the record must have its own field assignment procedure and its own field alteration clause. Generating all these functions in the basetype package is somewhat disconcerting when in all likelihood few, if any, of the functions will ever be called.

Other functions generated from templates are the valueerror function for the entire record, the aggregate assignment procedure, and the basetype typedescriptor and initial value declarations.

2.3.5 Base Types

Basetypes have been adequately discussed earlier in this section. It is worth noting once again that the compiler is quite basetype oriented, and that typedescriptors provide the most critical guide for type specific behavior in the program at runtime.

The compiler requires basetypes and typedescriptors to have been declared internally in order for most predefined operations to be performed. These declarations are made in conjunction user type declarations. Anonymous types, or types without names, create problems within the compiler. A user may create an anonymous type in one of two ways: 1) by declaring an object of a subrange type, which has already been addressed, and 2) by placing levels of anonymity in type declarations. Consider the type declaration:

```
type seqarray = array ([1..10]) of sequence (10) of integer;
```

The sequence type embedded in the definition is anonymous. An item selected from the array would have no named type.

Ada would not allow such anonymity in type declarations, and we feel that using them is a poor programming practice. It will be disallowed in Gypsy 2.1. Pending the implementation of 2.1, the compiler detects and disallows user specified anonymous types of the embedded variety.

2.4 Expressions

2.4.1 Name Expressions

Although Ada has same treatment of name expressions as Gypsy, the compiler’s abstraction of structured data types and their operations does not allow a normal treatment of structured object names in many places. In those places where normal treatment is sufficient, it is used with glee.

2.4.1.1 Component Selectors

Component selection on structured types is where name expressions demand unusual treatment. In particular, abstraction functions for field selection on structured types other than records must be used rather than direct access. (Direct access would mean normal indexing on arrays. It would mean a clumsy series of linked record accesses on dynamic types.) Also, special treatment is required for element selection on the left hand side of assignments. As explained in section 2.3, since access functions may not appear on a left hand side of assignments and since condition generation is different, basetype procedures are generated for element assignments. An indexerror on the left hand side is recognized in the appropriate assignment procedure for the object. Name expressions on structured types no longer appear, then, in units directly translated from user code. They are all protected in the abstraction functions for the various basetypes, where they make their only appearances.

2.4.2 Value Expressions

The concept of a value expression in Ada is exactly that of Gypsy.

2.4.2.1 Primary Values

The distinction the compiler makes between structured name and value (name) expressions is illustrated by the use of basetype-generated selector functions where value selectors were used in Gypsy. These are clearly illustrated in section 2.3.3.1.

Entry values (primed expressions) are intended for use in specification only. Their use in executable code will disallow the translation. In non-executable specifications, which become comments, they are translated in their original syntax.

Scalar literals are all declared in the basetype package. Therefore, to refer to literal "red" of type "color", it is necessary to use the qualified literal "basetype_package.color_basetype’(red)".

As previously stated, strings are implemented as arrays in Ada and as sequences in Gypsy. The compiler handles them as it would sequences. Translation of a string literal, then is done by coercion of the literal with a predefined function in GYPSY_PACKAGE which takes a string as an argument and returns a CHARACTER_SEQ (a predefined basetype, sequence of character). The string "A string" would translate to the function call:

```
gypsy_package.coerce_string ("A string")
```

Rational values are not translated in executable code. All other primary values are translated directly in to their Ada equivalents.

2.4.2.2 Modified Primary Values

Structure element alterations are performed with functions generated with the basetypes of each structured type. A subsequence selector function is provided in the sequence basetype package.

2.4.2.3 Operators

The treatment of specific operators has been extensively described in section 2.3 on types. Precedence levels are preserved in the Gypsy prefix, and emerge naturally from the compiler.

2.4.2.4 If Expression

No equivalent to the if expression construct exists in the Ada language. Translation of if expressions requires assignment to a temporary, which will serve as a place holder in the statement containing the expression when it is translated immediately afterward. Consider the translation of the statement:

```
a[i] := foo(if a[j] < a[k] then a[j] else a[k] fi);
```

where a is an array of integer values. In translating the statement we would first recognize the presence of an if expression and compute its value, assigning it to the temporary g_a_temp. Then g_a_temp would be substituted for the if expression in the translation of the statement. The generated Ada code would then be:

```
declare
  g_a_temp: integer := 0;
  begin
  if a[j] lt a[k] then g_a_temp := a[j]
    else g_a_temp := a[k]; end if;
  a[i] := foo(g_a_temp);
end;
```

G_a_temp is declared in the block to be a variable of the basetype of a[j], i.e. integer.

In the case of having more than one if expression embedded in the same Gypsy statement, we would simply have embedded blocks of code. Hence the rather baroque (and useless) Gypsy statement:

```
a[i] := if (if not j in 1..k then j in k..i else j in 1..i)
          then a[j] else a[k];

would translate to:

declare
  g_a_temp1: integer := 0;
  if (if not j in 1..k then j in k..i else j in 1..i)
    then g_a_temp1 := a[j]
    else g_a_temp1 := a[k];
  a[i] := g_a_temp1;
end;

which in turn expands to:

declare
  g_a_temp1: integer := 0;
  declare
    g_a_temp2: boolean := false;
    if not j in 1..k
      then g_a_temp2 := j in k..i;
      else g_a_temp2 := j in 1..i;
    if g_a_temp2 then g_a_temp1 := a[j];
                 else g_a_temp1 := a[k];
  end;
  a[i] := g_a_temp1;
end;
```

Gypsy semantics are such that evaluation of Gypsy expressions is free of side effects. This property is required to preserve the semantics of the if expression through this translation. It is sufficient for preservation of semantics, however, only in the absence of conditions and condition handling. Extracting the if expression from the context of a larger expression so that it can be pre-evaluated in an if statement can perturb the order in which conditions can be signalled. Preserving the correct order of evaluation under these circumstances is such an extensive problem that this initial implementation will not attempt to perform it.

2.4.3 Pre-computable Expressions

Pre-computability was not defined in Gypsy 2.0, on which the initial implementation of the compiler was based, and the Gypsy 2.0 parser placed very tight constraints on what expressions were regarded as pre-computable. Under these constraints, there is no danger that a pre-computable expression will not be pre-computable upon its translation into Ada.

Gypsy 2.1 liberalizes the previous constraints on pre-computability. Although it stops short of declaring exactly what expressions are pre- computable in a given implementation, it does define the set of expressions which are candidates for pre-computability. The standard documentation on Ada does not address the question of pre-computability, but it is hoped that any good Ada implementation will be able to pre-compute the value of any translated pre-computable Gypsy expression.

2.5 Programs

The Ada concept of program is compatible with the Gypsy notion. In particular, those Ada constructs which are targeted by the Gypsy translation are philosophically consistent with the Gypsy constructs, although they will vary in appearance. The effect of translated code on its environment is limited to changing the value of its data objects, raising an exception, and reading and writing files. The external environment specification for each program is provided by the implementor of the compiler for a particular target machine. Ada's concept of procedure, function, and constant is similar to that of Gypsy.

2.5.1 Procedures

Ada procedures are fundamentally quite similar to Gypsy procedures. Those aspects which differ are more appropriately discussed in other sections, notably those on textual organization and condition handling and in other subsections of this section on programs. User-defined Gypsy procedures are translated directly into Ada procedures with the same name. A return statement must be inserted at the end of the procedure statement list, since Gypsy procedures normally exit when they execute the last statement.

2.5.2 External Environment

Although the Ada concept of the external environment to a procedure is somewhat different from Gypsy, the net effect of translating a procedure into Ada preserves the semantics of the Gypsy procedure. The point of consistency between the environments of Ada and Gypsy procedures is that they are defined by a combination of those objects in the formal parameter lists and those externally visible at the point of declaration of the procedure.

Gypsy procedures have access only to those conditions supplied as actuals to the list of formal condition parameters. Ada procedures have no "exception parameters". Rather, they have access to those exceptions which are declared in any environment which nests the procedure at runtime. Visibility of external constants is defined by the environment of the scope or package which contains the procedure declaration.

While the external environment of an Ada procedure is somewhat different than that of a Gypsy procedure, symbols used in a translated procedure will naturally be only those which were used in the Gypsy procedure plus references to initial value constants and typedescriptors, to which the Ada procedure has access. Ambiguity among symbols in the external environment is eliminated by using global names for all external references. (Ex. The initial value for the basetype of an array type foo declared in scope bar would be basetype_package.foo_bar.initial_value.) Therefore the set of symbols referred to in the Ada procedure represents the same symbols referred to directly or inferred in the Gypsy procedure.

Gypsy "var" parameters to procedures are translated to "in out" parameters in the corresponding Ada procedure. "Const" parameters in Gypsy procedures become "in" parameters to Ada procedures.

2.5.3 Functions

The relationship between Gypsy and Ada functions and procedures is the same as between Gypsy and Ada procedures.

Ada has no item analogous to the predefined RESULT of Gypsy functions. Therefore RESULT is declared as a local variable in each translated Ada function. Its type is the type of the function, and its assigned initial value is the initial value for the type. When Gypsy functions terminate normally, the value of RESULT is returned. Ada functions must terminate normally with a RETURN statement which has a parameter, the value to be returned. The value supplied in the translated functions is the locally declared RESULT.

2.5.4 Constants

Ada constant declarations are exactly analogous to Gypsy constant declarations, except for the degree to which the concepts of pre-computability vary. This is implementation dependent in both languages, but the rather conservative rules on pre-computability, plus the fact that translated standard Gypsy functions are elaborated before any translated user declarations are given, seems to ensure that any acceptable constant declaration in Gypsy will translate directly into a declaration acceptable to Ada. The Gypsy parser will evaluate constant values, so the question of how to handle exceptions which arise during this evaluation is not of concern.

2.5.5 Bodies

Not all of the material which goes into the body of a Gypsy unit is meaningful to the program at runtime. In particular, non-validated specifications have no impact on execution and are converted into Ada comments in translation.

Pending bodies are not wholly acceptable to the compiler, since they reflect that the program translated is incompletely implemented and not ready to be run. If the body of a type or constant is pending, the compiler will abort. If a statement list is pending, the singleton statement list "raise routineerror;" is generated and a message is sent to the tty.

A special transformation is made on all user-declared function and procedure bodies to facilitate condition handling. This is discussed in Section 2.8.

2.5.6 Internal Environment

The internal environment of both Gypsy and Ada procedures may include the declaration of local variables, constants, and conditions (exceptions).

The only point of difference involves initial values for variables and constants. All Gypsy objects have initial values, which are either supplied explicitly by the user or assumed to be the default initial value of the type of the object. The internal objects of a Gypsy procedure are created in order of declaration and given their initial values explicitly. This means that the initial value of a symbol may depend on the initial value of any previously declared symbol. In addition, evaluation of the initial value of the local may result in the signalling of a condition. These conditions will propogate directly out of the routine, either through the condition parameter list or as routineerror. They cannot be locally handled by the user, since they are raised in a transitional state of the routine calling mechanism.

The valueerror which can result from the initial value assignment, however, must be treated specially. The initial value assignment in Ada is not an assignment treated consistently with normal assignment statements, since it cannot be replaced in-line with a call to a procedure which can check for valueerror. To perform the valueerror check, the compiler would have to put an explicit assignment statement at the beginning of the procedure for each variable declaration. Direct assignment in the declaration, however, would likely be more efficient, and is at least more closely resembles the Gypsy code.

The compiler attempts to use assignments in declarations whenever possible. It prints the symbol declarations in order. Whenever it finds a declaration where the user has specified an initial value, it attempts to determine if the initial value might generate a valueerror. If it sees there will be no valueerror, it will print the initialization in the declaration and treat succeeding symbols in the same manner. If it cannot determine that valueerror will not occur, it will print a declaration without an initialization and insert a call to the proper assignment procedure at the beginning of the statement list for the procedure. All subsequent declarations must also be made with an explicit call to the assignment procedure in order to preserve the correct order of evaluation. For example, consider the following local declarations in Gypsy (where type small_int = integer [1..10] and there is a formal parameter int_param of type integer):

```
var int1: small_int := 10;
var int2: small_int := intparam;
var int3: small_int := int2 - 1;
var int4: small_int;
```

These will translate to:

```
int1: small_int := 10;
int2: small_int;
int3: small_int;
int4: small_int := small_int_initial;
gypsy_package.integer_assign (int2, small_int_typedescriptor,
                              intparam);
gypsy_package.integer_assign (int3, small_int_typedescriptor,
                              int2 - 1);
```

Access specifications on internal objects, as with all objects, are ignored for the time being. When data abstraction is implemented in the compiler, access lists will be used to help steer determination of abstract equality, but its effect will be more or less invisible in the translated code.

2.5.7 Internal Statements

Of the various statement forms, only the if composition, case composition, loop composition, and begin composition survive intact. The procedure statement is used profusely, but user calls to procedures are embedded in blocks which perform valueerror and aliasing checks. All the others, save the cobegin statement (concurrency is not yet handled), are replaced with calls to procedures provided in the basetype support.

2.5.7.1 Assignment Statement

All assignment statements are converted into calls to the assignment procedure provided for the type of object on the left hand side. The parameters to the assignment procedure are some representation of the name expression on the left hand side, the typedescriptor appropriate for the name expression, and the expression on the right hand side. These procedures first check to see if the value on the right hand side is a legal value for the name expression (using the typedescriptor as a guide). If so, they raise a valueerror exception. Otherwise they make the assignment by invoking Ada assignment procedures.

The name expression on the left hand side may have a selector list with as many selectors as it takes to descend through a data structure to whichever component is to receive the expression on the right hand side. This creates a serious problem in Ada translation, since assignment statements for structured objects are generated as generic procedures, and these procedures must have the entire variable structure as a parameter. The only such procedures generated for structured types are the structure assignment and the component assignment (one level deep).

The only reasonable solution, then, is to decompose the assignment into assignments to temporaries one level at a time. For example:

```
type foo = sequence of integer;
```

```
  type bar = sequence of foo;
  ...
  var a: bar;
  a[i][j] := 10;
```

would translate into:

```
  var a: bar;
  declare
    var g_a_temp: foo;
    foo_basetype_assign (g_a_temp, a[i]);
    foo_basetype_element_assign (g_a_temp, j, 10);
    bar_basetype_element_assign (a, i, g_a_temp);
  end;
```

It is not difficult to see that this mechanism is clumsy and grossly inefficient at best, and becomes more so as the levels of descent into a structure on the left hand side increase. With structured types implemented as they are, however, it is the only way in which the semantics of structured component assignments may be maintained.

2.5.7.2 Input and Output

Input and output of Gypsy programs, as currently supported, is performed entirely through buffers. Those buffers which are parameters to the main program are designated as I/O buffers rather than normal internally used buffers. The send and receive operations on these buffers are the I/O operations print and read.

Ada input and output is performed through a predefined Ada package SEQUENTIAL_IO. This package must be instantiated for each different type of object which will be involved. Thus, if both characters and integers will be output, two package instantiations must occur, with each defining a set of input and output functions and file types which may be used with the appropriate data objects. The file types are IN_FILE, OUT_FILE, and INOUT_FILE for each type of object. "Internal" files are then declared to be of these file types in the same manner as any other declaration. The package instantiations and internal file declarations occur at the end of basetype_package. For example:

```
  -- Here we instantiate the I/O package for character types

  package character_io is new sequential_io
                                (element_type => character);

  -- This is a declaration of an "internal" file for output of
  -- characters

  char_file: character_io.inout_file;
```

To associate devices, or external files, with "internal" files, the CREATE, OPEN, and CLOSE procedures are invoked. These associations are made in the supermain routine which is the calling point for the compiled program. For example, the following statement will open the file whose device name is the string value of the variable FILENAME. This could, for instance, be "<cmp.akers>inchars.history".

```
  basetype_package.character_io.open
    (file => basetype_package.char_file, name => FILENAME);
```

An operation to write to the file would look like:

```
  character_io.write (file => basetype_package.char_file,
                      item => 'a');
```

A further description of the manner in which input\output objects are treated is given in the sections on the main program and implementation prelude.

Since Gypsy buffers may be passed around as parameters, the routines which call send and receive have no way of statically knowing whether they are performing input/output or internal message sending. Since the Ada input/output mechanism is different from the mechanism employed to perform buffer operations, some distinction will be needed at runtime in the Ada translation to determine what kind of operation to perform.

To this end an extra integer field will be included in the Ada image of Gypsy buffer objects. The field, named io_flag, will have the value 0 if the buffer is for internal use, and a positive value which associates it with one of the internal files if it is an input/output buffer. The value corresponds to the ordinal position of the buffer in the parameter list to the user's designated main procedure. A send or receive statement is translated into a case statement which keys on the io_flag field of the buffer. If the value is 0, a normal send to an internal buffer is performed. If it is non-zero, a write or read for the appropriate internal file is generated. Here is the Ada image of the Gypsy statement "send 'a to buf;":

```
if gypsy_package.character_valueerror
             (buftype_descriptor.b_elem_type, 'a')
        then raise valueerror;
 else case buf.io_flag is
       when 0 => basetype_package.buftype_test.send
                    ('a', buftype_descriptor, buf);
       when 1 => basetype_package.character_io.print
                    (basetype_package.char_file, 'a');
       when others => raise caseerror;
      end case;
 end if;
```

2.5.7.3 If Composition

The Gypsy if composition statement form is exactly the same as the one in Ada, excepting minor points of syntax, and is translated directly.

2.5.7.4 Case Composition

The Ada case composition statement is like that of Gypsy, except that the when others clause, corresponding to the else clause of the Gypsy case statement, is required. If there is no else clause in a Gypsy case composition and none of the case arms match the value of the label expression, a caseerror is signalled at runtime. In order to duplicate this behavior, an absent else clause is translated into an Ada when others clause which explicitly raises the caseerror exception.

2.5.7.5 Loop Composition

The loop statement in Ada is somewhat more general than the Gypsy loop composition, but it includes a syntactic and semantic form which exactly matches the Gypsy construct. The translation is direct.

2.5.8 Procedure and Function Calls

The Ada procedure and function calling mechanisms are fundamentally similar to Gypsy, although they are somewhat more flexible. To correctly handle procedure and function calls, however, the compiler must ensure that conditions which may be signalled from the call are generated and handled according to Gypsy rules. The solution to this problem is somewhat different for functions than for procedures.

Aside from conditions which propogate out of the called function, the only condition which may be signalled from a Gypsy function call is the valueerror which may be raised when an actual parameter violates the type constraints of its formal. In Ada this would result in a constraint_error (or in the case of translated dynamic types would likely go undetected). In either case this must be prevented. Since function calls may be embedded in arbitrarily complex expressions, it would be prohibitively complicated to check for valueerror at

the call site. Neither can the checks be made in the called function since Ada would have by that time detected the violation through its own calling mechanism.

To resolve this difficulty, each user defined function is declared with an interface function. This function is called rather than the original, or parent, function. Its formal parameters are the same as those of the parent, except that their types are the basetypes of the corresponding formals in the parent. The type of the function is the basetype of the type of the parent. The function checks each of its parameters for valueerror against the type of the corresponding parent formal, and raises the valueerror exception if it finds a violation. This exception will propagate back to the call site, producing the same behavior as Gypsy specifies. If there are no violations, the parent function is called, and its value returned as the value of the interface function. For example, consider the following function declaration:

```
function add1 (i: int): int =
 begin
  result := i + 1;
 end;
```

The function add1 is renamed to g_a_add1 and a new function add1 is defined as follows:

```
function add1 (i : in integer) return gypsy_package.int is
 result : gypsy_package.int := 0;
 begin
  if gypsy_package.integer_valueerror
              (gypsy_package.int_typedescriptor, i)
         then raise valueerror;
    end if;
  result := g_a_add1 (i);
  return result;
 end add1;

function g_a_add1 (i : in gypsy_package.int)
                    return gypsy_package.int is
 result : gypsy_package.int := 0;
 begin
  gypsy_package.integer_assign
       (result, gypsy_package.integer_plus (i, 1),
                gypsy_package.int_typedescriptor);
  return result;
 end g_a_add1;
```

If the types of the parameters to the user function are all basetypes, the interface function serves no purpose, since valueerror may not occur. In this case the interface function is not declared.

Valueerror checking must be provided on procedure calls also. Since the procedure call is a free-standing statement (as opposed to the function call, which may be embedded in a complex expression), the checks may be done at the call site without complication. The called procedure need not be affected by the translation of the calling mechanism. For example, consider the following procedure defined in scope s (as above):

```
procedure amin (var i: small_int; a: small_int_array) = ...
```

A call to amin of the form "amin (i_actual, a_actual);" would need the structure:

```
begin
 if gypsy_package.integer_valueerror (small_int_descriptor,
                                      i_actual)
   then raise valueerror; end if;
 if basetype_package.small_int_array_s.valueerror_occurs
       (small_int_array_descriptor, a_actual)
   then raise valueerror; end if;
 amin (i_actual, a_actual);
```

```
 end;
```

Checking for aliasing on var parameters, discussed later, may also be embedded in this structure.

There are two problems with this scheme of procedure calling. One is that if an actual parameter is an expression which must be evaluated to produce a value, the evaluation will be done twice: once in the call to the valueerror function and once in the call to the procedure itself. The other problem occurs where the actuals corresponding to var formals are name expressions which index into a structured item (Ex. a[i] or rec.field1). In such a case, the Ada representation of the actual is a function call (the function which performs indexing on objects of the basetype of the root of the actual). This violates the constraint that variable names must be used as actuals where the formal is an in or in out parameter.

In-line emulation of the value-result parameter passing scheme will solve both of these problems simultaneously. In this scheme, the value of the actual is calculated and assigned to a temporary of the type of the formal. The temporary is used as the actual parameter in the procedure call itself, and its value upon return is re-assigned to the variable from which it was created. Note that the expression is evaluated only when it is assigned to the temporary. It is checked for valueerror in the assignment, since the assignment will be performed by the assignment procedure for the basetype of the formal with the typedescriptor of the formal. Any conditions arising out of evaluation will arise in the proper order with respect to the rest of the computation. The temporary has a name, which satisfies the constraint for actuals of var parameters. The reassignment to the original actual will be performed with the typedescriptor for the original actual, so valueerror will be checked correctly and at the appropriate time. Incidentally, the checks for aliaserror may be inserted cleanly between the assignments to temporaries and the call to the procedure, so that any conditions arising from that check will appear at the appropriate time.

Consider this Gypsy example, particularly the call to proc:

```
scope bim = begin
  type foo = array ([1..10]) of integer;
  procedure proc (var i1: int; i2: int) = pending;
  procedure caller (var arr: foo; i,j: integer) =
   begin
    proc (arr[i], arr[j]);
   end;
end;
```

The call "proc (arr[i], arr[j])" becomes the block statement:

```
declare
 g_a_temp1, g_a_temp2 : gypsy_package.int;
 begin
  gypsy_package.integer_assign (g_a_temp1,
     basetype_package.foo_bim.select_component (arr, i),
     gypsy_package.int_typedescriptor);
  gypsy_package.integer_assign (g_a_temp2,
     basetype_package.foo_bim.select_component (arr, j),
     gypsy_package.int_typedescriptor);
  if i = j then raise aliaserror; end if;
  proc (g_a_temp1, g_a_temp2);
  basetype_package.foo_bim.element_assign
      (arr, foo_descriptor, i, g_a_temp1);
 end;
```

2.5.8.1 Actual Parameters

The treatment of actual parameters is embedded in the previous discussion. The runtime varerror check which was required in Gypsy 2.0 is no longer necessary in Gypsy 2.1, due to the disallowance of local declarations whose types are restricted by the values of formal parameters.

2.5.8.2 Type Consistency

Type consistency has been thoroughly discussed above.

2.5.8.3 Aliasing

Some alias checking, specifically the check to see if a candidate pair of actual parameters are exactly the same name expression, may be performed by the parser. The rest of the check must be performed at runtime. The compiler must check for pairs of type-compatible formal parameters where at least one is a variable parameter and the roots of both name expressions are the same. In this case, the indexes into each expression must be checked at runtime for equality. If they are equal down to the finest index of the shallowest name expression (a.b[1] is shallower than a.b[1].c.), aliasing exists and aliaserror must be signalled. Consider the procedure header and call below:

```
type int_array = array ([1..10]) of integer;
type array1 = array ([1..10]) of int_array;
type seq1 = sequence of array1;
procedure proc (var i,j: integer) =  ...
procedure foo (var s1, s2: seq1) = begin
 var i1, i2, i3, i4, i5, i6: integer;
 ...
 proc (s1[i1,i2,i3], s2[i4,i5,i6]);
```

For the sake of clarity in the example, let us consider that the valueerror checks have been optimized away and that for the moment we need not be concerned with the conversion of the translated actual parameter expressions into name expressions. The aliaserror check will be included in the procedure calling mechanism in the following manner:

```
if (i1 eq i4) and (i2 eq i5) and (i3 eq i6)
  then raise aliaserror
  else proc (s1[i1,i2,i3],s2[i4,i5,i6]);
```

This statement will be inserted in the procedure call block immediately preceding the call itself. (See also the last example in Section 2.5.8.)

2.5.8.4 Transfer of Control

The code to transfer control through a procedure call is created so as to preserve the order of operations described in the Gypsy manual. In particular, the aliaserror check will be made after the valueerror check and may occur either before or after the creation of name expressions to use as actual parameters.

2.5.9 Getting Started

2.5.9.1 Verification Environment

The generation of Ada code from Gypsy code is directed entirely from within the Gypsy Verification Environment. The exec command in the environment which orders a translation is the translate command. The command is menu driven by the exec grammar.

The translate command initiates an interactive dialogue with the user to acquire necessary information. An annotated dialogue may be found in Appendix E.

2.5.9.2 Main Program

When the user invokes the compiler, he supplies the name of an implementation prelude to serve as the environment for the execution of his program and a call to a procedure which is interpreted to be the main program. The main program specified must be a Gypsy procedure. As currently supported, only buffers are allowed to be variable parameters to the main procedure. These buffers are treated specially throughout the Gypsy program. (See the section on input and output.) The types of all parameters to the main program must have their basetype defined in the given implementation prelude. In the user's call to main, the actual parameters are names chosen from the "variables" in the chosen environment. These variables must be type-compatible with the formals in the user's main procedure. Note that in the prelude, the range restriction constants are "pending", which means in this case that they will be supplied by the compiler to match those of the formals in the user's main procedure.

The main procedure is transformed into three separate procedures which are all placed in the same package as the user's main. If the user's main were named, for instance, TOP_PROC, the compiler would produce procedures named G_A_SUPERMAIN, G_A_TOP_PROC_ENTRY, and TOP_PROC. TOP_PROC itself is not treated specially in the translation. Its header, however, is used as the basis for the generation of an intermediate procedure, G_A_TOP_PROC_ENTRY, which sets the io_flag fields of the buffer parameters and does valueerror checking before calling TOP_PROC. G_A_SUPERMAIN is the outermost procedure, and it serves to link the program with its execution environment and determine the system-level devices which are being used by the program.

Consider the following procedure which is designated as the main program.

```
procedure top_proc (var b: buftype) = begin
bufproc (b);
```