Proving Gypsy Programs

February 9, 1989
Technical Report #4-b
supercedes May, 1986 edition

Richard M. Cohen

Revised by Robert L. Akers

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This research was supported in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be intepreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the National Computer Security Center, or the U.S. Government.

Abstract

Program verification applies formal understanding of programming language semantics to the practical problem of constructing reliable software. The programming language Gypsy and the Gypsy Verification Environment, a programming environment that supports development of programs, specifications, and proofs, represent significant steps in making the program verification technology available for practical use.

The purpose of this dissertation is to specify clearly the meaning of Gypsy programs, and the means by which they are proven. The semantics are presented in a "semi-formal" way, so that they are more accessible to the verification practitioner, than would be more formal, mathematical semantics. First an operational semantics for a subset of Gypsy is presented. Then this model is extended to cover more complex features of the language. Using the execution semantics as the underlying conceptual base, we proceed to present the mechanisms used to generate verification conditions for the full Gypsy language, including data abstraction, concurrent programming, and exception handling.

Acknowledgments

I owe many debts to my friends and colleagues for their support and encouragement throughout this work. None of this work would have been done without the interest and enthusiasm of Don Good, the primary force behind the creation of Gypsy. Bob Boyer was instrumental in helping me avoid several blind alleys, and convincing me I was done. Dan Craigen read an early draft of my dissertation, and his comments helped me improve upon it.

My past and present colleagues at the Institute for Computing Science have contributed to my efforts in many ways. They have provided a good environment for discussion, elaboration, and refinement of my understanding of Gypsy. They have also provided friendship and intellectual interest that has helped me through these efforts.

Many friends have encouraged me throughout my graduate career, and without them I surely would not have persevered. For this I acknowledge special gratitude to my dear friends Dave and Paula Matuszek, John McHugh, Ruth Baldwin, Paul Reynolds, and Bernice Borak. The members of C.E.R.F. offered encouragement, and I trust they are satisfied with the result. Many others have been helpful, and I do not mean to slight them by not mentioning their names. It only indicates that I have a poor memory.

And lastly, I must acknowledge the support of my wife, Dorene. She not only put up with me when I was tired, depressed, excited, and frequently absent, but remained supportive and enthusiastic. We have supported each other through our studies financially, emotionally, and spiritually. Neither of us would have finished without the other's support.

Chapter 1
INTRODUCTION

Gypsy is a program description language. That is, Gypsy is a language for describing computer programs and their specifications. This report defines the proof methods for the language Gypsy. A simple semantic model of a language feature is first presented and then extended as required to encompass the additional complexities of the full language. A verification condition generator based on this model is then developed. The elaboration is pedagogical and is meant to provide an understanding of the meaning of Gypsy programs. The goal of this report is to provide the reader with this understanding.

The presentation describes the unique proof methods for handling concurrency and data abstraction in Gypsy. Gypsy supports concurrent programming using message buffers and supports specification of concurrent programs by defining buffer histories, records of the messages sent to and received from a buffer.

The reader is expected to be familiar with the Gypsy language, as described by the Gypsy 2.1 language report [Good 85].[1] Footnotes generally discuss details that relate the current topic to topics that are discussed in later sections. They may be skipped on first reading.

1.1 Why Gypsy is Important and What I Have Done

I have defined the semantics of the Gypsy language and the formal methods used to prove the consistency of Gypsy programs and specifications. Gypsy and the Gypsy Verification Environment (GVE) mark a significant step in the development of practical methods for program verification. The development of these proof methods form the underlying basis for Gypsy verification. Moreover, the definition style used here (mapping an operational definition into a verification condition generator) offers an intuitive basis for the verification by clearly relating an operational understanding of the program to the verification conditions (VCs) required for the proof. This reinforces in the programmer's mind the relation between a procedural understanding of the program (as is common to most programmers) and the requirements of the proof methods encountered during the verification.

Gypsy proof methods include two areas often left out of formal semantic definitions:

- concurrency, and
- exception handling.

[1] The reader who is familiar with the Gypsy 2.0 language report [Good 78a] should not find the differences particularly bothersome.

The proof methods developed for concurrent programs in Gypsy were the first formalization of the semantics of message buffers. Earlier descriptions and examples of the Gypsy concurrency have already been published in [Good 78b] and [Good 79].

1.2 Semantic Definition Method

The programming language Gypsy and informal proof methods for Gypsy programs have evolved over several years. An important goal in this evolutionary design has been modularity of programs and their proofs. Modularity in programs is traditionally enforced by abstraction and information hiding. An acceptable semantic definition of Gypsy must reflect those mechanisms into the proof methods.

Normally denotational semantic definitions embody no abstraction. The denotation of a function declaration is the mathematical function it computes. References to the function from other points in the program evoke that denotation and hence, complete knowledge of its definition. This violates the requirement of information hiding.

To solve this we take a two-step approach, presenting:

1. an execution model which lacks abstraction, and then
2. methods (derived from the execution model) to generate and prove the verification conditions (VCs) derived from the program. The VCs and their proofs will be constructed to permit proof to reflect abstraction in the program.

The primitive execution model is presented mainly to provide an intuitive base on which to develop the VC generator. The simple execution model is unacceptable in our proofs, as it violates our requirement of modularity. The VC generator supports our notions of modular and abstract proofs, but masks the full semantic interpretation of program units by reducing the usable knowledge of a routine to its (possibly incomplete) external specifications.

In practice, the formal specifications of verified software often specify less than full functionality, stating only the critical program properties. Thus, programs that meet their formal specifications may not exhibit other required (but not formally specified) behavior. (E.g., real-time behavior is ignored in most formal specification languages.)

1.3 Nondeterminism

Nondeterminism arises for many reasons and is necessary for most distributed computing systems. This nondeterminism causes significant complications in programming. Thus, program verification is of particular interest for distributed or parallel systems, as they are much harder to understand than sequential programs. This same nondeterminism also complicates programming language semantics, causes significant complications in program verification, and interferes with the use of most mechanical theorem provers. Most theorem proving work is done in theories with deterministic functions. If F is a nondeterministic function, then $F(X) = F(X)$ may not be a valid formula.

Our semantics allow us to manage nondeterministic program constructs. Basically, the extensions add contextual knowledge about otherwise uninterpreted function symbols. This allows us to reason about the function symbols without revealing complete details. Use of the "context" mechanism is sufficient to provide a weak definition of concurrent Gypsy constructs. We use the abstraction facility to hide many of the nondeterministic details of how the concurrent processes interact. The Gypsy specification language does not allow us to describe the full behavior of Gypsy concurrent processes. Certain aspects of concurrent program behavior that are evident in the intuitive operational model cannot be expressed in the specification language. The semantics of concurrency given here are incomplete in exactly the same way. The modularity and abstraction that form the basis of the *Independence Principle* [Good 85, p. 2] are enforced equally in the programming language and in the specification language. Gypsy buffers resemble shared variables. There is no mechanism to pass a buffer "exclusively" to a procedure, indicating that no other active procedure has access to that buffer. Therefore, the independence principle forces us to treat the buffer parameter as a shared buffer, which other procedures may manipulate while this procedure is active. With sufficient global knowledge of the calling structure program, it might be possible to determine that no other procedures could in fact access the buffer while this procedure was active; such knowledge cannot be used in a proof of the Gypsy procedure, as it would violate the independence principle.

Gypsy procedures may be nondeterministic. For example, in a concurrent procedure call, the result may depend on the order in which events occur in the parallel procedure invocations. The semantics of the concurrent procedure call do not specify the order of the events. Any ordering information regarding the events must be gained from the specification abstraction of the called procedures.

1.4 Incremental Development of Programs and Proofs

Programs are usually developed by iterative refinement and rewriting. The direction may be top-down or bottom-up, but some backtracking is almost always necessary. Later, programs are often revised and extended (an activity called "maintenance"). Modifying a program may require recompilation. It may also require modifications to the proof. Constructing formal specifications and proofs is very time consuming and labor intensive. Redoing the proof of an entire program would be extremely expensive, and if verification is to be practical, small modifications in a program must require only small modifications in the proof. The same notions of abstraction that allow us to modify programs easily also allow us to construct modular proofs. The portion of a proof that depends on a particular piece of the program should be comparable to the portion of the program that depends on that piece. The VC generation semantics allow us to reflect the abstraction mechanisms built into the language into the proof methods, and hence, to support modularity in proofs just as the language supports modularity in the program. This allows the GVE's proof manager [Moriconi 77] to handle incremental changes during program development.

1.5 Negotiable Semantics

The language definer has a choice between defining rigorous, inflexible semantics, leaving no aspect of program behavior undefined, or defining looser, flexible semantics, leaving some aspects of program behavior unspecified. (The appearance of the word "undefined" in language definitions usually indicates presence of the latter approach.) Some have argued as to whether the presence of "undefined" in a language definition is good or bad [Palme 74]. One main reason for introducing these unspecified portions into the language definition is to provide some leeway for the implementor. Some details of a language feature may be irrelevant or tangential to the intended behavior of a program. Leaving these details of the language unspecified may allow the use of a variety of implementation techniques. Conversely, specifying every detail of the language may preclude some desirable implementation options.

Some semantic definition techniques lend themselves to leaving some things undefined, while others do not. Often the semantic definitional methods make it easier or cleaner for the language definer to completely specify program behavior. In the semantic definition of Gypsy, the order of argument evaluation is specified, and call-by-value/call-by-value-result semantics are used to describe program behavior.[2] This admits several reasonable operational models, which are simpler than ones involving call-by-reference. However, this also has the effect of requiring Gypsy compilers to preserve these semantics even when attempting to use the more efficient call-by-reference technique at run time. We propose a language extension that allows the programmer to "request" relaxation of some details of the language definition. The formal semantics do not reflect any leniency in the formally required program behavior, but the proof methods do recognize this relaxation request and relax the semantics used to verify that the program conforms to its specifications. Thus, the programmer can explicitly annotate the program to indicate that some details for the program behavior are not required. The proof methods will use the relaxed program semantics in verifying the program, and an optimizing compiler might be allowed to take advantage of the relaxed semantics to apply specific optimizations.

The "negotiation" allows the program writer to specify an abstract equality relation on program states, masking certain uninteresting distinctions. Based on this, weaker proof rules can be used and a possible explosion of cases can be avoided.

This "relaxation" of semantics is particularly helpful in the case of exception conditions in Gypsy programs. Gypsy programs typically handle many exception conditions with a single "condition handler", and it is often not necessary to deal with the detailed state information that distinguishes one particular condition from another. Thus, if the programmer could specify that the values of some variables were not of interest when an exception condition is raised, then the proofs relating to the exception condition would not have to distinguish among the multiple signalling states.

1.6 Outline of Chapters

1.6.1 Introduction

The introduction provides a brief overview of the work and presentation, and tries to gently introduce the reader to the issues at hand.

1.6.2 Related Work

This chapter discusses the nature of program verification, and identifies several tutorial works on verification. Other mechanical systems supporting program verification are described. Several other efforts have addressed the areas of specifying and proving concurrent programs and exception handling. These are contrasted with the approach presented here.

1.6.3 Some Remarks about Gypsy Programs

This chapter deals with several issues underlying the Gypsy proof methods. The first section outlines the properties of the Gypsy language that helped to make the semantic definition simple. Gypsy does not distinguish function declarations intended for use in specifications from those intended for execution. Thus, Gypsy functions have meaning in program proofs, as well as a possible interpretation as an executable function. The nature of this dual definition is discussed, as well as several aspects of Gypsy

[2] The Gypsy 2.1 Report [Good 85] defines assignment and other basic operations of the language (including function invocation) in terms of procedures and procedure calls. The approach taken here is slightly different. We use assignment and function invocation as primitive operations, and reduce the other parts of the language (including procedure invocation) to these two primitives. This different approach was chosen to reduce the number of primitives required in the semantic description, but is logically equivalent to that taken in the language report.

specifications and their analysis.

1.6.4 Normalizing Gypsy Programs

Many Gypsy constructs can be eliminated by treating them as abbreviations with expansions composed from a smaller base language. In order to simplify the semantic definition, the number of constructs in the base language is reduced. The normalization process reduces Gypsy programs to six executable statements and three specification statements. The details of normalization and expanding syntactic abbreviations are described.

1.6.5 A Simple Operational Model

A simple operational model for Gypsy programs is presented. The model is first developed without considering exception handling or concurrency. The model is then extended to include exception handling. Concurrency is dealt with in Chapter 8.

1.6.6 A Verification Condition Generator

Based on the operational model in Chapter 5, a verification condition generator (VC generator) is described. A forward VC generation technique is used to provide a close relation between the operational model and the VC generation process. The VC generator first analyzes the control flow of the routine to compute a set of linear control path segments, and then symbolically executes each path segment to generate the VCs. Methods for proving termination of execution of sequential Gypsy programs are also described.

1.6.7 Buffers and Activation_ids

The basic Gypsy concurrency mechanism is based on message buffers. Specification of concurrent programs is based on histories of the messages sent to and received from buffers. The history mechanism uses a unique activation_id to identify each dynamic routine invocation, in order to pair each message in the history with the specific routine activation that performed the corresponding send or receive operation on the buffer.

1.6.8 Concurrency

The Gypsy concurrent procedure call (**`COBEGIN`** statement) is a generalization of the sequential procedure call. The basis of specifying and verifying concurrent Gypsy programs is to describe the effect of the **`COBEGIN`** on buffer parameters. The **`BLOCK`** specification is a technique for describing behavior of non-terminating concurrent programs. The interpretation of the **`BLOCK`** specification is given, along with its proof methods.

1.6.9 Proof of Data Abstraction

Data abstraction does not affect the execution semantics of the program. Data abstraction merely imposes visibility restrictions on the program description and proof. Program code and specifications dealing with the abstract data type (not its implementation) must be independent of the details of the data representation. External specifications of a routine include both abstract specifications, which are visible everywhere, and concrete specifications, which are only visible in contexts in which the data representation is known. These two sets of specifications must be proven to be consistent. Gypsy allows the programmer to define equality on abstract data types. To allow the normal rules of substitution in proofs, it is necessary to show that the user has defined an equivalence relation, and that all functions defined on the abstract data type behave deterministically under this equivalence relation.

1.6.10 Proposed Language Changes

In pursuing this work, several relatively minor changes came to light that might improve the language. The most significant of these are the suggested extensions to support proof of well-definedness of specification functions and termination of (sequential) executable routines.

1.6.11 Future Work

The work presented suggests several lines for future research, such as the application of this style of semantic definition to other languages, and more formal models of VC generation. Several other topics for further work are discussed.

1.7 Lacuna: Topics Omitted

The precise version of the language addressed here lies partly between Gypsy 2.0 and Gypsy 2.1. The language is basically Gypsy 2.1 as described in the July 1985 draft language manual [Good 85]. This work proceeded in parallel with the evolution of Gypsy 2.1 from Gypsy 2.0, and some aspects of this "moving target" may yet be evident in the manuscript. Some of the language changes incorporated in Gypsy 2.1 were based on these efforts to clarify and formalize the semantics of Gypsy.

The static semantics or compile-time type consistency requirements of Gypsy are not discussed here, nor are the semantics of Gypsy data structures. The type consistency rules are adequately explained in the Gypsy 2.0 and Gypsy 2.1 language reports. Gypsy's data structures are quite ordinary. They include integers, rationals, arrays, and records, all of which are common in many programming languages, and quite well understood. Gypsy also includes sets, sequences (indexed lists), and "mappings" (sets of ordered pairs), which are not present in most programming languages. But these less common data types are defined to behave as in common mathematical usage, and so they, too, are not of special interest here.

Chapter 2
RELATED WORK

2.1 Specification & Verification Techniques

There are several different aspects of "program correctness."

- Partial correctness deals with proving properties of the results of programs. The results are guaranteed to satisfy the properties *if the program terminates and produces any results.*
- Proof of termination addresses the question of whether a given program will terminate. Of course, we know that proof of termination is not possible for all programs, since this is precisely the Halting Problem [Hopcroft & Ullman 69].
- Total correctness is the combination of partial correctness plus a proof of termination.
- Safety properties are the analog of partial correctness in the world of concurrent programs. These deal with properties of partial results of running programs.
- Liveness properties and the absence of deadlock are the analogs of termination in the world of concurrent programs. These deal with the progress of a concurrent computation and its ultimate production of some result or action.

Many techniques have been proposed for specifying and proving partial correctness of programs (e.g., state machines, algebraic axioms, program assertions for specifications, and inductive assertions, structural induction, computational induction for proofs). The classic technique for proving partial correctness properties of sequential, procedural code is the Floyd-Hoare inductive assertion method [Floyd 67, Hoare 69]. This forms the basis for Gypsy verification, as it does for much of the practical verification work being done.

There are several introductions to the area of formal program verification, including [Manna 74, Hantler 76, Anderson 79, Boyer&Moore 81a]. [Berg 82] is particularly encyclopedic in its coverage of specification and verification techniques. Many progressive introductory programming texts, such as [Wirth 73], introduce the notions of formal specification and proof along with the more traditional notions of algorithmic programming. Some perspectives on program verification can be obtained by reading [London 75, Gries 78]. [Gordon 79] provides a very nice introduction to denotational semantics, and [Pagan 81] and [Tennent 81] provide a general introduction to programming language semantics.

2.2 Verification Systems

Interest in the formal correctness of programs dates from the 1960's [McCarthy 63, Floyd 67, Dijkstra 68, Hoare 69]. Elspas *et al.* suggested that mechanical theorem provers were required to make program verification practical as early as 1972 [Elspas 72]. Yet Lipton *et al.* seem to have misunderstood the importance of mechanical proof checkers in 1979 [De Millo 79].

King [King 69] was the first to implement a mechanical program verifier based on Floyd's work. His verifier used backward substitution to generate VCs for a simple Algol-like language manipulating integers.

The Stanford Pascal Verifier [von Henke 75, Luckham 79] is perhaps best known for Wolfgang Polak's proof of a Pascal compiler [Polak 80]. However, the user must supply the theorem prover with a large number of axioms in order to complete the proofs, and it is extremely difficult for humans to construct large sets of axioms without introducing a contradiction.

AFFIRM [Gerhart 80, Thompson 81, Erickson 81] started as a verifier for abstract data types based on the Knuth-Bendix algorithm [Knuth 69] for testing the completeness of a system of rewrite rules. The system was extended to support VC generation for Pascal-like programs, although the proofs about abstract data types (including inductive proofs) remains AFFIRM's strength.

The Hierarchical Development Methodology (HDM) was developed at Stanford Research Institute, later SRI International [Robinson 75, Robinson 77, Robinson 79, Silverberg 79, Levitt 79]. It centered around the program specification language SPECIAL [Roubine 77], which uses a state machine model to describe changes to a global machine state (i.e., global variables), based on the work of David Parnas [Parnas 72a, Parnas 72b]. The user then proves properties of the initial machine state and invariants about reachable states. Lower level machines may be defined, and mappings between the higher level and lower level machines may be shown to preserve the verified properties of the higher level machine. Ultimately, the state machine specification is used as a specification for a program written in a standard programming language. HDM adherents claim that this "programming language independence" is an advantage. However, HDM only proves properties of a non-procedural program specification, which must then be interpreted by a human programmer in order to produce an actual program. Thus, the relevance of the specification proof to the actual program depends on the integrity of the human programmer.

A new version of HDM is under development [Crow 85a]. Revised SPECIAL is based on Hoare-axioms [Crow 85b], and the new HDM system accepts statements about fragments of Pascal programs using Hoare's notation. Larger program fragments can then be built from smaller ones by the rule of composition, until full Pascal programs have been constructed.

Ina Jo [Locasso 80] is the specification language component of the Formal Development Methodology developed at Systems Development Corporation. Ina Jo is another state-machine oriented program specification language. It is mated with a low-level interactive theorem prover called ITP [Schorre 84]. As with HDM, this system only proves properties of non-procedural state-machine program specifications. An attempt is being made to extend Ina Jo to support specification of concurrent programs using temporal logic [Nixon 85].

Starting in the early 1970's, Boyer and Moore built one of the most impressive mechanical theorem provers in existence [Boyer&Moore 75, Boyer&Moore 79]. They use structural induction to prove properties of pure LISP programs, using LISP as the specification language as well as the "programming language." They implemented a VC generator for a large subset of FORTRAN in order to produce more theorems for their prover to prove [Boyer&Moore 81b].

The Gypsy Verification Environment has evolved over a period of ten years. It's predecessor, jointly developed by USC Information Sciences Institute and The University of Texas, is described in [Good 75]. Various aspects of the GVE are described in [Moriconi 77, Hare 79, Smith 80, Akers 83]. Some of the notable successes of the Gypsy effort are described in [DiVito 81, DiVito 82, Good 82, Siebert 84, Good 84].

2.3 Proof of Concurrent Programs

Owicki and Gries [Owicki 75, Owicki 76] handle concurrent programs with globally shared variables. The only constraint is that the underlying machine provide the uninterruptable memory operations read-a-word and write-a-word. Their method first proves properties of the individual, sequential procedures, and then shows that there is no "interference" between the operation of the concurrent procedures and the assumptions of the individual proofs. Thus, the sequential proofs remain valid for the concurrent program. While this is a very powerful technique, the non-interference proofs can grow very large.

Apt *et al.* [Apt 80] developed an axiomatic proof technique for Hoare's CSP. They deal with both safety properties and freedom from deadlock. This requires proving "cooperation" of the component, sequential proofs, which corresponds to global reasoning about the cooperating processes, in addition to the sequential proofs. The proofs of cooperation can grow as $O(n^2)$ in the length of the program, as is the case with Owicki and Gries.

Levin has also developed an axiomatic proof technique for CSP [Levin 81]. He uses shared auxiliary variables, rather than global invariants as Apt *et al.* Thus, in constructing the proof of a program from those of its sequential processes, he must prove both a "satisfaction" proof, that all possible message exchanges justify the post-condition following a send or receive, and a non-interference proof (similar to Owicki and Gries). He views communication between processes as a means to an end, rather than as the purpose of the computation. Thus, it is natural to use auxiliary variables to reflect as much or as little of the communication history as needed.

Monitors [Hoare 74] provide a mechanism for structuring control of and access to shared objects. Proof methods for monitors were introduced [Howard 76] as Gypsy 1.0 was being designed. Indeed, monitors were originally envisioned as the concurrency mechanism to be included in Gypsy. However, the examples that were driving the initial concurrent verification work at that time were mainly communication and networking problems, and early examples of Gypsy programs tended to use monitors to build bounded buffers. This observation led to the adoption of message buffers as a simpler, more constrained concurrency mechanism, and ultimately led to simpler proof methods.

2.4 Exception Handling

The Gypsy mechanisms for handling exception conditions is based on the work of Goodenough [Goodenough 75]. A significant extension added in Gypsy is the mapping of condition names at routine boundaries, in order to preserve the independence principle.

A condition signalled in a routine is propagated into the calling environment as **`ROUTINEERROR`** if the condition signalled is neither handled within the routine nor is it a condition parameter of the routine.[3] Propagating the original condition unchanged would leave unconstrained the possible exception conditions that could arise at a call site, depending on all of the routines in the possible call-tree below that point. There would be no way to analyze the possible control paths due to exceptions without global knowledge of all of the routines involved. Mapping the condition names at routine boundaries allows analysis to be done based only on the local procedure and the external specifications of called routines. Luckham's work on exception handling in Ada™ has used the Gypsy **EXIT CASE** style of specification in Anna. [Luckham 80]

Optimizing compilers prove theorems about programs in order to justify that optimizing transformations do not alter the semantics of a program. Typically, either the optimizers simply fail to do the proofs, or apply optimizations for which the proofs can be done very simply (perhaps due to the language design). McHugh used clean termination proofs of Gypsy statements to support optimization of Gypsy programs [McHugh 83]. Typical optimizations involving code motion and the suppression of run-time tests preserve program semantics only in the absence of run-time errors. By using a description of the run-time environment, McHugh was able to prove the absence of certain run-time errors, and hence apply program optimizations safely.

Several attempts have been made to extend algebraic specifications to describe operations that may encounter run-time error conditions. The notion of an extended algebra including error terms and undefined terms is discussed in [Majster 79] and [Musser 77].

Euclid [Lampson 77]was initially designed at the same time as Gypsy, but was intended more to support practical programming efforts than program verification research. Still, verifiability was an important goal of the Euclid work. Following the style of the axiomatic definition of Pascal done by Hoare and Wirth [Hoare 73], proof rules were developed for Euclid [London 78], and a verifier for a close descendent of Euclid [Crowe 82] is currently under construction [Bonyun 82, Craigen 84]. Euclid took an interesting approach to exception handling. No legal Euclid program can signal an exception. In order to be a legal Euclid program, various "legality assertions" must be proven, showing that run-time errors will not occur. Thus, one cannot determine whether a program is a legal Euclid program without doing the proofs! Early work on proving absense of run-time errors was done by Richard Sites [Sites 74]. Sites developed techniques for proving clean termination of flow-graph programs. Later, Steve German worked on proving absence of run-time errors in Pascal programs [German 81]. German produced a working implementation based on his formalism.

2.5 Key Results of the Gypsy Project

The Gypsy Project has addressed the breadth of program verification from language design and formal semantics, to mechanical tools for supporting proofs about real programs, to compilers for the language. Here are some of the key results of these efforts.

- Integration of specifications & program into a single language.
- Use of message buffers as concurrency mechanism, and first proof methods for buffers. The simple structure of concurrent programs has allowed proofs of significant, real-world, distributed programs -- verified Gypsy programs have actually run on distributed hardware!

---

[3] The one exception to this rule is that **SPACEERROR** propagates unchanged. This is because **SPACEERROR** represents violation of an implementation defined constraint on computing resources, a resource limit not directly under the control of the invoked routine.

- The Independence Principle -- an enforcement of modularity in program code, specifications, and proof -- allows incremental program development (top-down, or bottom-up) with manageable consequences to the verification.
- Incremental development manager
    - manages ripple effects of changes during program development and verification,
    - only invalidates proofs that use changed elements of the program,
    - provides a verification *environment* that integrates program development, specification, and verification, along with editing and compilation.
- Interactive theorem prover to decouple automatic proving technology from practice of verification. The prover helps in discovering proofs (the hard part, the intellectual challenge), but should not stand in the way of the user completing the rest of the verification if a single formula cannot be proven.
- Forward VC generation & natural deduction prover help the programmer/verifier maintain context during development.[4]

[4] [Good 70] is one of the first descriptions of "forward accumulation" in VC generation, although this was not the major reason for choosing forward VC generation for the GVE.

Chapter 3

SOME REMARKS ABOUT GYPSY PROGRAMS

3.1 Why the Semantics of Gypsy are Easier than Many Other Languages

3.1.1 Parameter Passing Mechanisms

Gypsy outlaws the uses of variables that commonly cause problems in programming and are awkward in semantic definitions. The following common programming language sore spots are illegal:

- dangerous aliasing (overlapping parameter references to **VAR** parameters)
- general shared variables (shared between concurrent processes)
- undetermined values resulting from routines that exit abnormally (e.g., after a run-time arithmetic error).

In Gypsy programs the call-by-value-result and call-by-reference mechanisms for passing data parameters are equivalent.[5] Banning dangerous aliasing for data parameters and returning well-defined values when routines terminate abnormally eliminate the circumstances in which these parameter passing mechanisms are distinguishable for data parameters.[6] Buffer parameters represent a restricted case of shared variables, and the no-aliasing rules are slightly relaxed. Thus, slightly more complexity is required to deal with passing buffer parameters within a concurrent procedure call.

3.1.2 No Dangerous Aliasing

No dangerous aliasing is *ever* allowed within the argument list of a single procedure. This means there is no need for explicit use of *locations* (see, for example, [Stoy 77], or [Tennent 81]) in the semantics of Gypsy.

Locations are an elegant mechanism for modeling:

- explicit storage allocation and deallocation,
- call-by-reference parameter passing,
- aliasing in parameter passage, and

---

[5] This is not the case for buffer parameters.

[6] The Gypsy 2.0 condition **VARERROR** has been eliminated in Gypsy 2.1. Type consistency restrictions in Gypsy 2.1 assure that the assignment to the formal parameters required by "call by value" can be performed properly if the actual data parameters yield proper values.

- dangling references.

Locations allow this by introducing a level of indirection in resolving all identifier references. This mechanism is useful when we want to include a certain level of implementation details in the semantic definition, or when the programming language in question requires this more cumbersome mechanism. We choose not to use this mechanism, and Gypsy does not require it.

3.1.3 Simple Control Structures

Gypsy does not have a **`GO TO`** statement. Further, all control structures must always be exited "properly." It is impossible for a statement to abort the execution of an encompassing control structure. (Signalling a condition does not abort encompassing control structures -- the encompassing control structures may handle the condition, or, in the case of routines, may map the condition name into some other condition name.) Thus, evaluation of Gypsy programs follows the conventional stack model of expression evaluation (augmented to handle conditions).

Consequently, there is no need to use the cumbersome (but fully general and powerful) mechanism of *continuations* to describe Gypsy control structures. (See, for example, [Milne 76], or [Gordon 79].) Simple function composition is sufficient. We do not need to introduce the explicit notions of an interpreter and control stack. The basic notion of function evaluation is enough.

3.1.4 Procedure Calls

Since no non-local variable references exist within Gypsy procedures, procedures cannot refer to global variables. Similarly, since no **`GO TO`** statements exist in Gypsy, no global labels can be referenced. Further, there are no non-local exits from the procedure body during execution of a procedure call. In addition, the signalling of error conditions involves the normal (proper) unwinding of the calling stack. Thus, the proper "post-lude" of the procedure is always executed, and no "pre-mature block exit" is possible.

3.1.5 Run-Time Error Conditions

Gypsy defines a mechanism for handling exception conditions that arise during execution of a program. There is no exception handling defined for specifications, and the remainder of this section deals only with executable functions. During execution, all routine invocations must either:

- terminate normally,
- terminate abnormally (i.e., signalling an exception condition), or
- fail to terminate.

In effect, all executable functions and procedures that terminate, either normally or abnormally, under all circumstances, are total functions. There is no need to worry about clean termination of executable functions. (Routines are defined to return an exception condition if they do terminate abnormally. Thus, all routines return values composed of a condition name -- perhaps **`*NORMAL`** -- and the function result or procedure output parameters. The execution-time notion which occurs in some programming languages, of a function yielding an undefined result, does not occur in Gypsy. A function that terminates abnormally simply yields a value with the condition name being something other than **`*NORMAL`**.)

Restrictions on the proper domain of a function can appear as "type restrictions" on the formal parameters, or as predicates in the entry specification. Routine invocations need not terminate normally simply because the actual parameters are type consistent with the formal parameter list, and the routine's entry specification is satisfied. In fact, functions that never terminate normally are perfectly acceptable, as for example,

```
function Never (x:integer) : boolean unless (cond C) =
  begin
    entry x in [1..10];

    signal C;
  end;
```

This function always terminates abnormally, signalling its condition parameter. This declaration denotes a function that is defined over all integers if we acknowledge that functions return an ordered pair (a condition and a normal value).[7] In this case, **`Never`** would return the pair **`(C, FALSE)`**, where **`C`** stands for the function's actual condition parameter, and **`FALSE`** is the default initial value for type boolean.

The only way that a sequential routine may fail to terminate and yield a value is to execute a non-terminating **`LOOP`** statement, or invoke a routine that does not terminate. Gypsy **`SEND`** and **`RECEIVE`** statements are modeled as procedure calls, so blocking for input or output also fits this model.

3.2 Data Abstraction

Data abstraction is merely an enforced programmer discipline and has no effect on dynamic semantics. The discipline forces the programmer to avoid having Gypsy programs and proofs depend on the concrete representation of data types.

In Gypsy, abstract data type declarations contain a list of routine names, identifying the routines inside which the concrete data representation is visible. The visibility restrictions affect type checking and accessing of the concrete representation in specifications and executable code. Within routines that have concrete access to an abstract type, the abstract type is considered type equivalent to the concrete type. Outside those routines, the two types are not considered equivalent.

Visibility restrictions affect the proof methods implemented in the Gypsy Verification Environment. Thus, proof dependencies are restricted to reflect the dependencies permitted in routines. The proof of a routine lacking concrete access to an abstract data type, just as the executable code of that routine, cannot directly depend on the concrete structure of that data type.

Later in the discussion on generation of verification conditions, we see that function names are left uninterpreted. Information about the behavior of the function is introduced when external specifications of the routine are brought into the proof.

3.3 Duality of Function Definitions

[7] It would be totally undefined if we only considered the normal value portion of the result.

3.3.1 Specification Functions and Executable Functions

Function declarations in Gypsy include the following aspects:

- executable code (i.e., input to the compiler to generate machine executable code),
- internal and external specifications (i.e., input intended for purposes of formally verifying properties of the program)
- type specifications (which are used for both compilation and proof purposes).

Within this Gypsy function declaration there are actually *two* function definitions -- one executable, the other a specification. These are distinct and separate (though related) function definitions. *The main goal of verification of Gypsy programs is to prove that the executable function conforms to the specification function.* That is, both yield the same value whenever both are well-defined. We say that the executable function is a *restriction* of the specification function. For convenience, we consider a function to be a restriction of itself.

The entry and exit specifications[8] define a function (in a purely mathematical domain). Conjuncts in function exit specifications can be divided into two classes: function definitions (i.e., conjuncts of the form **`RESULT = ...`**, and similar forms), and everything else. Any conjunct appearing in a function exit specification that is not a function definition can be written as a Gypsy lemma.[9] Hence, we will assume that exit specifications contain only function definitions.

The executable body of the function declaration defines the function that is to be evaluated when the program runs.[10] These two definitions are intended to be closely related to one another -- that is precisely the role of the Gypsy proof methods.

> *It is a key concept in composing and verifying Gypsy programs that these two function definitions are distinct, and that they serve very different roles within the Gypsy proof methods.*

Specification function refers to the function defined in the exit specification. ***Executable function*** refers to the function defined in the executable body of the declaration.

3.3.2 Specification Functions and Specification Abstraction

Specification functions are the basic Gypsy mechanism for proof and specification abstraction. Abstraction plays a parallel role in program implementation, specification, and proof. Just as procedural abstraction is used to suppress implementation details in the executable program, the abstraction of the specification function is used to suppress details of the implementation and specification in the proof of the program. In both cases, the object is to build an implementation or proof structure that is modular; an internal modification to one function should have only minor, local effects through the rest of the program or proof. This property has been recognized as a key in program implementation (e.g., see the literature on structured programming, top-down design, and step-wise refinement) and has been identified as a key step toward practical program verification.

---

[8]The following remarks discuss only abstract specifications. These comments mostly apply to concrete specifications, except that Gypsy lemmas can only refer to abstract properties.

[9]Actually, definitions could be written as lemmas, too, but the proof methods for proving that the executable function conforms to the specification function is keyed to the exit specification.

[10]A particular Gypsy implementation may have constraints that cause its execution of a given Gypsy program to evaluate some function that is a restriction of the function defined by the executable body. That is, the implemented executable function may signal **`ROUTINEERROR`** or **`SPACEERROR`** under some circumstances under which the executable body would not.

3.3.3 The Relation Between the Two Functions

In our proofs and specifications, we use the specification function defined by the function declaration *in place of* the executable function. All Gypsy specifications (e.g., **ASSERT** statements, **ENTRY** and **EXIT** specifications) use the function name to refer to the specification function. Thus, by requiring that the executable function be a restriction of the specification function, properties proven using the specification function will hold for the executable function when it terminates normally.

Specifications may describe program behavior on a set larger than the proper domain[11] of the executable function. If so, our proofs will include consideration of cases that do not correspond to any possible program execution, as well as all the cases that do correspond to possible program executions. Thus, it is important that the specification function be a suitable stand-in for our implementation in each reference. To assure this, we must prove the following:

- that the executable function conforms to the specification function (i.e., the executable function, which may be a partial function, is a restriction of the specification function),
- that the specification function is a well-defined function over some specified domain, and
- that in our specifications and proof each use of the specification function applies the function to actual parameters in its proper domain.[12]

An additional property is desirable (in terms of keeping our specifications and proofs closely related to the expected behavior of our program):

- that the executable function be well-defined (i.e., terminate normally) over some identified domain.

This last property allows us to restrict use of the specification function to instances that correspond to possible uses of the executable function. If we omit this restriction, then we are free to extend our specification to describe extensions to program behavior (and hence prove theorems about behavior) that cannot be realized by the executable program. Nothing is wrong with proving these possibly interesting theorems, but it is desirable that the proof methods help us to understand when we have drifted from proving things about our executable programs and proceed to prove properties of our specification abstractions.

3.3.4 Domains of the Two Functions

The ***proper domain*** of a function refers to a set of values over which the function terminates normally and returns a proper value (i.e., not bottom, the undefined element, and -- in the case of an executable function -- not an abnormal condition).

The specification function definition given in the exit specification should probably be a total function. For the time being we will relax this view, and only require that the specification function be well-defined when the entry condition for the function declaration is satisfied. Thus, the proper domain for the specification function is identified by the entry specification. The proper domain for the executable

[11]The proper domain of an executable function is the set of input parameter values for which the function terminates normally and returns a proper value.

[12]The current GVE only proves these last two points for the Gypsy predefined functions, not for user defined functions. A future enhancement of the GVE will remove this potential for constructing unsound proofs.

function may be smaller than that of the specification function.[13] In this case, the executable body will signal an abnormal termination condition for some input values that satisfy the entry specification. Our proof methods must assure that whenever the executable function terminates normally and yields a value, the value is precisely that of the specification function definition. In other words, the executable function must be a restriction of the specification function (i.e., they must agree wherever both are defined, and the proper domain of the executable function must be a subset of the proper domain of the specification function). Further, if we prove the proper domain of the executable function is a subset of the proper domain of the exit specification, then we can assume the well-definedness of the exit specification in any context corresponding to normal termination of the executable function.

3.4 Prescriptive versus Descriptive Specifications

We divide specifications of routines into two classes: *prescriptive* specifications and *descriptive* specifications. A prescriptive specification lays down a rule, and dictates that certain actions must be taken when computing the function. A descriptive specification expresses qualities or properties of the function, serving to describe necessary conditions when certain actions are taken. This is a particularly important distinction when defining functions that may terminate abnormally. Prescriptive specifications require that the routine signal an exception condition when a certain precondition is met (or not met). Descriptive specifications specify what the normal result of the computation should be (without regard to the preconditions of any particular implementation of the computation), and describe conditions that hold if an exception condition is signalled. Thus, the descriptive specifications permit an implementation to compute a correct result whenever possible, specifying a predicate that holds when an exception condition is signalled. A prescriptive specification gives a precondition that defines a set of values outside the proper domain of the function; a descriptive specification gives a characterization of a superset of values outside the domain of the function.

Below we define **foo** prescriptively, and **bar** descriptively.

```
         {Prescriptive version:  }

function foo (x:T) unless (cond abnormal_cond)
   begin
     entry prescriptive_predicate(x)
              otherwise abnormal_cond;
     exit  foo(x) = alter_x(x);
     result:=alter_x(x) unless (abnormal_cond);
   end;

         {Descriptive version:  }

function bar (x:T) unless (cond abnormal_cond)
   begin
     entry true;
     exit case (is normal:        bar(x) = alter_x(x);
                is abnormal_cond:
                            descriptive_predicate(x));
     result:=alter_x(x) unless (abnormal_cond);
   end;

function alter_x (x:T) unless (abnormal_cond) = pending;
```

[13] It can even be larger, but we are not interested in any input values not admitted by the entry specification, because they are neither relevant to our proof, nor to the expected program behavior at execution time.

Foo is not allowed to return a normal result outside the set characterized by **prescriptive_predicate**, while **BAR** is not allowed to signal an exception condition outside the set characterized by **descriptive_predicate**. Thus, in verifying **BAR**, we would have to establish that **descriptive_predicate** holds whenever **Alter_x** terminates abnormally. This is obviously required, since **BAR** terminates abnormally precisely when **Alter_x** does.[14]

3.5 Relation of Proofs to Real Implementations

All implementations of Gypsy implement only subsets of full Gypsy language; they are required to compute restrictions of the functions defined or described by the formal meaning of the program (i.e., implemented functions can signal conditions more often than the formally-defined executable function, but must always agree when both are well defined). In practice, implementations are likely to "punt" and signal an abnormal condition when computations require arithmetic precision greater than the hardware word size, or when a program requires more memory than is physically present. Such behavior is permitted by this semantic definition, if the appropriate abnormal condition is signalled.

The mechanism for proving absence of run-time errors discussed in [McHugh 83] allows us to talk about clean termination of a Gypsy program on real hardware, rather than only on an ideal Gypsy machine.

3.6 Nondeterminism in Gypsy

3.6.1 What is nondeterminism, and why do we have it?

A routine is nondeterministic if its result (or output) is not a well-defined mathematical function of its input. A mathematical equality function is required to be symmetric, transitive, and reflexive. Since we do not know that **f(x) = f(x)** when **f** is a nondeterministic function, we would have to restrict very severely our use of equality, and much of our ability to apply normal mathematical theorem proving and algebraic techniques. To avoid this, Gypsy specifications functions are constrained to be deterministic.[15]

Nondeterminism in programming languages can arise for several reasons.

- imprecise language definitions
- precise language definitions that intentionally leave open some implementation choices (usually in the interest of efficiency or ease of implementation)
- languages intended for writing algorithms at a "high level" -- not allowing the programmer to specify "low level" (and presumably unimportant) details of the program
- data abstraction (i.e., functions may appear deterministic at the abstract level, but not at the concrete level. Consider a function that performs a table look up, and as a side effect reorders the table to speed up future queries, preserving abstract equality on the table, but not concrete equality.)
- concurrency (either due to nondeterministic parallel computations within a routine, or due to interaction with external concurrent computation).

---

[14]Implementations can signal implementation error conditions, such as memory size limits, which can further restrict the actual domain of a function. In this case, exception conditions, such as stack overflow, can arise at execution time, causing **BAR** to terminate abnormally, even though **Alter_x** is normally defined for those arguments. See section 3.5, page 19.

[15]And the Gypsy verification system requires that specification functions be proven to be deterministic.

3.6.2 Sources of Nondeterminism in Gypsy

Gypsy procedures are permitted to behave nondeterministically. The Gypsy 2.1 language has three sources of nondeterministic behavior.[16]

a. Explicit Concurrency. The **`COBEGIN`** statement, which supports possible parallel execution of program parts, does not specify any timing constraints on the order of events in different arms of the **`COBEGIN`**. Three functions defined on buffers (**`FULL`**, **`EMPTY`**, and **`CONTENT`**) can reveal properties of buffers as truly shared objects. Thus, these functions may behave nondeterministically.

b. Nondeterministic Buffer Polling. The **`AWAIT`** statement allows a sequential Gypsy procedure to wait until any one of several buffer operations can complete. For example, a procedure could thus respond to input from any one of several input buffers, using the **`AWAIT`** to suspend its execution until any one of the buffers became non-empty. This is essentially a nondeterministic **`CASE`** statement in which the case arms are labeled with buffer operations. The nondeterminism is at the statement level, and does not threaten the determinacy of our expression language.

c. Data Abstraction. Procedures in which the concrete representation of an abstract data type is visible can reveal information about that concrete representation. Thus, the procedures may not behave deterministically from an abstract point of view. Functions with concrete access to the data type are not permitted to violate the visibility rules of the abstract data type.[17]

3.6.2-A Apparently Nondeterministic Procedure Calls

Procedure invocation (as opposed to function invocation) need not appear deterministic. Thus, results are permitted to depend on external events or scheduling decisions. Procedure results can differ by violating data abstraction rules (e.g., a procedure can produce different results when called with two abstract objects that appear to be equal at the abstract level).

3.6.2-B Resource Errors and Determinacy

Any routine invocation can terminate abnormally if for some reason the processor cannot complete the computation properly. Abnormal termination is reflected in the calling environment by signalling a condition. When a function invocation terminates abnormally, it does not yield a normal result value to be used in further computations. Our requirement for determinacy is restricted to the function invocations that yield normal values, since only these are in our specifications and proofs. Signalling an abnormal condition is reflected in the procedural execution of a routine, and the process of generating verification conditions maps this procedural behavior into the functional domain of verification conditions.

---

[16]Gypsy 2.0 had an additional source of nondeterminacy, expression evaluation. Within a Gypsy 2.0 expression, the language specifies that arguments (or nested expressions) are evaluated before higher level (or containing) expressions. There are two degrees of freedom in the order of evaluation:

i. of evaluation of actual parameters in a routine call, and

ii. of evaluation of arguments to an infix operator (e.g., "+").

(Actually, these are the same, as infix operators are merely syntactic abbreviations for function calls to predefined functions. Thus, the arguments are actual parameters in a routine call.) The flexibility permitted the implementor is *only* in the choice of which of several possible exception conditions to signal. If the expression yields a normal value, then all Gypsy implementations must produce the same result. *For this reason Gypsy 2.1 specifies the order of evaluation of expressions and function arguments.*

[17]All concrete access functions must be shown to behave deterministically under substitution using abstract equality. That is, the function must yield results that are equal under an appropriate equality function when it is applied to two sets of actual parameters that satisfy the abstract equality function associated with the abstract data type. See Chapter 9 for more details.

3.6.2-C Concurrent Routine Calls

The order of execution in a concurrent routine call (**`COBEGIN`** statement) is unspecified to allow the parallel routines to operate asynchronously. Sometimes their execution order will be determined by explicit external communication (i.e., outside the routine and its descendants); sometimes it will be determined by implicit external communication (e.g., a clock-driven scheduler); sometimes it will be determined by explicit internal communication between the concurrent routines. Often the system behavior will be determined by some combination of these circumstances.

3.6.2-D Expression Evaluation

Expression evaluation always yields a well-defined value, or a well-defined exception condition. Actual parameters in routine call statements are evaluated from left to right. If evaluation of one of the actual parameters terminates abnormally, then that condition is propagated.

The normal result of a Gypsy function must be determined by the actual parameters of the function call. The determinacy of the normal exit case may require proof that routine calls evaluated within the evaluation of the function are themselves deterministic, and that uses of data abstraction are sound. Abnormal termination conditions need not be deterministic, since abnormal termination can depend (for example) on implementation resource constraints.

The language constraints on expression evaluation derive from the following two goals:

- assuring that program execution can be accurately modeled by the algebra of the proof system,

and within that,

- allowing the implementor maximum flexibility.

In general, specifications of the predefined routines in Gypsy are of the form

If the routine exits normally, then the result is exactly ....

The language does not specify exactly when a routine will exit normally. Often the specification will not even specify when a routine must not exit normally. The main and pervasive constraint is that *if* the routine exits normally, then the result *must* be algebraically correct.

Further, every attempt is made in the language definition to be as machine independent as possible. Thus, while implementation constraints (such as integer overflow) are acknowledged as possible exit conditions for predefined routines (such as add), the circumstances under which these error conditions will be signalled are defined in terms of function definitions *supplied by the implementor*. The language definition is independent of what a particular implementation can properly evaluate. Any implementation is a restriction of the full language. A completely degenerate implementation, which cannot properly evaluate anything, might reduce all programs to signalling **`ROUTINEERROR`**. (That is, an implementor can always punt on any feature. Of course, most implementations attempt to support at least *some* useful programs.)

The language definition, particularly with the constraints on order of evaluation, requires the implementation to signal an appropriate error condition at a relatively well-defined time. The time at which the condition must be signalled is constrained by the order of observable side effects. Normally, this would mean that the signal must occur during expression evaluation, before the next side effect is performed. (In fact, the requirement is that the program state be consistent with the statically observable state from the original program, given the point of the signal. The implementor is free to evaluate pieces of the program in arbitrary order, only if the observable program state, order of observable side effects, and routine results are correct. The implementor is quite free to let incorrect computations proceed, only if the program state is rolled back appropriately before the error handler is entered.)

Evaluation of actual parameters includes three checks:

1. Actual parameters are handled in left-to-right order, each parameter being evaluated and then value checked to assure that it is an element in the value space of the formal parameter type.
2. If evaluation of any of the actual parameters does not yield a value, but terminates abnormally, then the exception condition signalled by the left-most parameter that failed to yield a value is signalled.
3. After all actual values have been determined to be proper, then all parameters are checked for aliasing, and **aliaserror** is signalled if any potentially dangerous aliasing is detected.[18]

We can model this by using explicit temporaries for function parameters (i.e., parameter evaluation becomes an assignment to a temporary, and then only simple variables are passed as parameters).[19]

3.6.3 An Aside on Optimization

In some sense these constraints on expression evaluation severely hamper attempts to optimize Gypsy programs. Gypsy statements such as

```
N := (1 + LargestRepresentableInteger) -
   LargestRepresentableInteger
```

appear to be *required* to signal **ADDERROR**. This follows because the subexpression

```
1 + LargestRepresentableInteger
```

cannot be evaluated, and so *must* signal an error condition, even though the value of the containing expression can easily be computed by an implementation with an intelligent optimizer.

The requirement that the expression

```
1 + LargestRepresentableInteger
```

must signal **ADDERROR** arises because the definition of the predefined routine **ADD** cannot examine the context of the routine call. Its behavior must be completely specified by its actual parameters. However, the optimizer can notice that the original expression

```
(1 + LargestRepresentableInteger) -
   LargestRepresentableInteger
```

is equivalent to

```
1 + (LargestRepresentableInteger -
   LargestRepresentableInteger),
```

---

[18]See Sections 5.2 and 5.3.1 for more discussion of aliasing restrictions.

[19]Evaluating all the actual parameters before checking for dangerous aliasing simplifies the modeling of parameter passing in routine calls.

which is algebraically equal to 1. This optimization is really a program transformation. The original program is transformed into a new, algebraically equivalent program. One difference between the programs is that some computations have been eliminated. It happens that one of the computations eliminated could not have been properly completed by the implementation, and thus the required signalling of an exception condition is eliminated. The semantics of the optimized program are precisely the same as those of the original program *on an ideal Gypsy machine*.[20]

Accordingly, we see that simple optimizing transformations cannot be applied without regard to possible error conditions. This suggests the following two possible courses for implementation of Gypsy optimizers:

- verify that no error signals are introduced or eliminated by optimizing transformations, or
- perform the optimizations at the Gypsy source program level, and allow the user to verify the new program against user specifications. (This course requires care, as user specifications in Gypsy programs are usually incomplete, and so simply preserving conformance with formal specifications is not sufficient.)

John McHugh's dissertation [McHugh 83] discusses this and related issues in verification driven program optimization.

3.6.4 Dealing with Nondeterminism in the Language Definition

In order to make execution of function calls deterministic, we will define the order of evaluation of actual parameters. Functions with concrete access to an abstract data type are forbidden to introduce nondeterminacy by revealing the underlying data representation.

We will deal with the nondeterminism in procedures introduced by concurrency through the abstraction mechanism introduced in chapters 7 and 8.

In order for proofs to be valid, functions are required to be deterministic. Determinacy follows almost directly from easy syntactic checks, though some proof obligation may be imposed. To demonstrate determinacy of a function definition, it is sufficient to show that:

- All called routines are deterministic[21], and
- The function neither invokes concurrent routine calls (via **COBEGIN**) nor awaits buffer operations (via **AWAIT**).

Functions cannot communicate with external processes through shared buffers. Functions do not accept **VAR** parameters, and so cannot communicate with external concurrent processes through shared buffers.[22] Determinacy of functions should be checked by the Gypsy Verification Environment, as part of the general process of proving properties of Gypsy programs.

---

[20] An ideal Gypsy machine is one in which there are no size or space restrictions. Arbitrarily large integers, sequences, sets, and mappings can be represented. **SPACEERROR** never occurs, nor do any of the exceptions from predefined functions that are based on finite precision arithmetic.

[21] The Gypsy 2.1 manual describes the language in terms of predefined procedures. This definition uses an equivalent paradigm that reduces the language to function calls plus assignment. Basic operations that the Gypsy 2.1 manual describes as predefined procedures are modeled as function calls plus assignment here. Thus, there are no procedures that need to be used in defining the semantics of a user defined function.

[22] The functions **FULL**, **EMPTY**, and **CONTENT** behave deterministically in the absence of concurrency and shared buffers.

Chapter 4
NORMALIZING GYPSY PROGRAMS

To simplify specifying the semantics of our programs, we will reduce certain Gypsy constructs to simpler forms. This allows us to define the semantics of a smaller language, and use these reductions to extend the definition to the larger language. These reductions come about in two places: mapping the full language onto a base language subset, and then "normalizing" the resulting program.

4.1 New Statement Forms

In order to reflect the various interactions between the procedural, executable portion of Gypsy and the proof mechanism, we introduce three new statement forms.

- **`ASSUME <BOOLEAN EXP>`**: denotes that **`<BOOLEAN EXP>`** should be assumed to hold at that point in the routine.
- **`PROVE <BOOLEAN EXP>`**: denotes that **`<BOOLEAN EXP>`** must be proven to hold at that point in the routine.
- **`BREAKPATH`**: indicates locations at which user-supplied assertions were placed. During VC generation these locations will be used to decompose the routine body into linear code segments.

These three statements can be understood by looking at the expansion of the normal Gypsy **`ASSERT`** statement. The statement

```
ASSERT prove Q
```

becomes

```
PROVE Q;
BREAKPATH;
ASSUME Q.
```

Here the three uses of the **`ASSERT`** are made explicit. It requires that **`Q`** be proven at that location, that this location is used to break the control paths into finite linear program segments, and that **`Q`** is to be assumed at the beginning of the next linear segment. The simpler case is the statement:

```
ASSERT (ASSUME p);
```

which becomes the canonical form:

```
ASSUME p;
```

The path is not broken, and no proof is required.

The **ASSERT** statements can also be expanded to include the statement that the routine's **CONST** parameters have not changed, and that they are equal to their initial values within the routine. In Gypsy the initial value of a formal parameter **X** is denoted by **X'**. Thus, **X = X'** holds for all formal data parameters at the beginning of a routine, and continues to hold for all **CONST** parameters throughout the routine. This automatic extension of the assertion relieves the programmer from having to repeat this statement again and again.

We also introduce one new statement form to subsume the place of <STATEMENT LIST> in the syntax. The statement form

```
PROG2 Statement1; Statement2 END
```

is a compound statement limited to exactly two statements. This is analogous to the function **PROG2** in LISP 1.5 [McCarthy 65]. **PROG2...END** is used to replace compound statements purely as a syntactic simplification, so that our rules have a fixed number of arguments.

4.2 Extended Body of a Routine

Portions of a Gypsy routine that are relevant to VC generation are sometimes elided, abbreviated, or placed "inconveniently" for describing VC generation in simple syntactic terms. For example, the **EXIT** specification is included as part of the routine header (along with the other externally visible specifications), rather than at the end of the executable body, where it might be thought to belong "logically." The *extended body* of a routine is an extension of the basic routine body (i.e., it's executable code body) formed by explicitly including the additional information from its external and internal specifications.[23] The following are the operations performed to construct the extended body of a routine:

1. The **KEEP** specification, an invariant for local routine variables and parameters, is placed before the beginning of the routine body as an explicit **PROVE**, and after every "relevant" statement in the routine body. A statement is "relevant" to the **KEEP** if it changes the value of a variable mentioned in the **KEEP** assertion.

2. Local variable initialization is moved from implicit assignments in local declarations to become explicit assignments at the beginning of the executable code body.[24]

3. The **ENTRY** specification is moved to the beginning of the extended body, and becomes an **ASSUME**.[25]

4. The **EXIT** specification is moved to the end of the extended body, and becomes a **PROVE** or **ASSUME** statement as appropriate. If the **EXIT** specification includes specification of possible abnormal exits, then the **EXIT** expansion wraps condition handlers around the routine body, with a condition handler for each condition mentioned in the **EXIT** specification explicitly containing the appropriate assertion from the **EXIT**.

An example makes this transformation clearer. The body of the procedure

---

[23]The notion of an extended routine body is also used to simplify program analysis in the Gypsy Optimizer [McHugh 83].

[24]The order of the declarations must be preserved in the order of the explicit assignments, so that any run-time exceptions would be signalled in the same way.

[25]In-line insertion of the concrete external specifications (the **CENTRY** and **CEXIT**) and concrete data invariants (the **HOLD** specification) is also performed similarly. For details of how data abstraction interacts with proofs, see chapter 9.

```
procedure P (var X:T1; const Y:T2) =
  begin
    entry  P_Entry(X, Y);
    exit   P_Exit(X, Y);

    var I:T3 := initial_I;
    keep   P_keep(x,y,i)

    routine_body(x,y,i);
  end;
```

expands into

```
X := X';                 {Initially these two identifiers denote}
Y := Y';                 {   the same values                    }
ASSUME P_Entry(X, Y);

I := initial_I;          {Assign initial values to local variables}
PROVE  P_keep(X,Y,I);    {Prove KEEP assertion for initial values }
ASSUME P_keep(X,Y,I);

routine_body(X,Y,I);

PROVE  P_keep(X,Y,I);    {if any of X, Y, or I was changed by     }
ASSUME P_keep(X,Y,I);    {     routine_body                       }

PROVE  P_Exit(X,Y);
BREAKPATH
```

If the **EXIT** specification is a **CASE** construct, then each arm of the **EXIT** specification is embedded in one arm of a condition handler surrounding the entire body of the routine. Each arm of the condition handler must also resignal the condition if it is to propagate into the calling environment. For example, the **EXIT** specification of

```
procedure P (var X:T1; const Y:T2) unless (cond C1, C2) =
  begin
    entry  P_Entry(X, Y);
    exit   CASE (IS c1:     Q1(X, Y);
                 IS c2:     Q2(X, Y);
                 IS NORMAL: Q3(X, Y));
    ...
  end;
```

would result in an extended body of the form:

```
begin
  ...
when is c1:         PROVE Q1(X, Y); signal c1;
     is c2:         PROVE Q2(X, Y); signal c2;
     is NORMAL:     PROVE Q3(X, Y);
     is SPACEERROR: signal SPACEERROR;
     else           signal ROUTINEERROR;
end;
```

The **else** clause at the end of the **when** shows that any condition signalled within the routine body other than **c1**, **c2** (the routine's condition parameters), or the predefined condition **SPACEERROR** will be propagated as the condition **ROUTINEERROR** in the calling environment.

4.3 Reductions to the Base Language

A number of surface level syntactic constructs disappear in the normalized representation of Gypsy programs. These reductions generally deal with optional parts of a construct, or with abbreviated forms permitted in the program.

4.3.1 Index Simplification

Component selectors of the form **`(i, j, ...)`** are an abbreviation for the component selectors **`(i) (j)`** .... Any of the index selectors in this expansion may be a subsequence selector (i.e., a range rather than simply an expression). For example, **`A[i,j]`** expands into **`A[i][j]`**.

4.3.2 CASE Statement

`CASE` statements are reduced to an **`IF...THEN...ELIF`** structure. A case statement of the form

```
CASE exp
  IS case1: stmt1
  IS case2: stmt2
    ...
  IS casen: stmtn
  ELSE      stmtx
END
```

expands into

```
IF   exp = case1 THEN stmt1
ELIF exp = case2 THEN stmt2
 ...
ELIF exp = casen THEN stmtn
ELSE                  stmtx
END
```

4.3.3 ELIF Statement

This syntactic form disappears from the base language. **`IF`** statements containing **`ELIF`**'s are all reduced to nested **`IF...THEN...ELSE...END`** forms. A **`IF`** statement of the form

```
IF   bool1 THEN stmt1
ELIF bool2 THEN stmt2
 ...
ELIF booln THEN stmtn
ELSE            stmtx
END
```

expands into

```
IF bool1
   THEN stmt1
   ELSE IF bool2
           THEN stmt2
           ELSE  ...

               ...IF booln
                      THEN stmtn
                      ELSE stmtx
                   END
               ...
        END
END
```

4.3.4 ASSERT

In order to separate the proof-time assertions from constructs with execution-time effects, we expand Gypsy **ASSERT** statements into distinct proof-time assertions, and execution-time control statements.

The statement

```
ASSERT P otherwise C
```

is taken as an abbreviation for

```
ASSERT P;
if not P then signal C end.
```

Run-time validation can always be expressed by the **if ... then signal ...** construct. This applies to validated routine entry specifications in the extended routine body, as well.

Run-time validation of assertions to be proven is also possible. Note that this amounts to run-time validation of the proof, or the assumptions of the proof. Such a run-time check can fail if the basis of the proof (e.g., a properly functioning Gypsy machine) is contradicted by the physical hardware.

Having dealt with run-time validation, we can now perform the canonicalization of **ASSERT** as described in Section 4.1.

4.3.5 Alteration Clauses

Alteration clauses are an expression notation for describing piecewise modifications to the value of a structured object. The notation is simply a different syntactic representation of calls to overloaded predefined Gypsy functions. Some of these functions are defined generically, others, specifically for array and record alterations, are defined as a consequent to definition of a new array or record type.[26]

The following transformations illustrate the mapping of alteration clause syntax into alteration function calls. Those functions which are consequentially defined contain a "*t*" in their names, that notation being a placeholder for the name of the type from which they were derived.

The form for modifying an existing element of a sequence or mapping,

---

[26]In the case of the array, the consequent function does array bounds checking, based on information about the index type wired into its defintion. For record alterations, there is one function defined for modifying each record field; the function has the name of the field wired into its definition.

```
x WITH ((y) := z)
```

is a representation of

```
gypsy_alter_element# (x, y, z)
```

The form for modifying an array element,

```
x WITH ((y) := z)
```

is a representation of

```
gypsy_alter_*t*_element# (x, y, z)
```

There is one such function for each field of a record type, so that

```
r WITH (.f := x)
```

is a representation of

```
gypsy_alter_*t*_record_field_f# (r, x)
```

The mapping alteration

```
x WITH (INTO (y) := z)
```

is a representation of

```
gypsy_alter_into_mapping_element# (x, y, z)
```

The sequence insertion alteration

```
x WITH (BEFORE (y) := z)
```

is a representation of

```
gypsy_alter_before_seq_element# (x, y, z)
```

Similarly,

```
x WITH (BEHIND (y) := z)
```

is a representation of

```
gypsy_alter_behind# (x, y, z)
```

Sequence element deletion

```
x WITH (SEQOMIT (y) := z)
```

is a representation of

```
gypsy_alter_seqomit# (x, y, z)
```

Similarly for mappings,

```
x WITH (MAPOMIT (y) := z)
```

is a representation of

```
gypsy_alter_mapomit# (x, y, z)
```

4.3.6 Assignment to Elements of Structures

Assignments to elements of arrays, sequences, records, and mappings are abbreviations for assignments of new values to the entire structure. These new values are described by Gypsy alteration clauses. For example, the Gypsy statements

```
A[i] := x;      {array}
R.F  := y;      {record}
S[i] := z;      {sequence}
M[x] := y;      {mapping}
```

are abbreviations for the statements

```
A := A with ([i] := x);
R := R with (.F  := y);
S := S with ([i] := z);
M := M with ([x] := y);
```

Multiple component selectors on the left-hand side of the assignment are treated recursively from right-to-left. That is, the name expressions **A**, **R**, **S**, and **M** above may contain further component selectors, in which case the expansions are applied repeatedly until the left-hand side of the assignment is simply an identifier. Sequence assignment statements using **before** and **behind** translate directly to alteration clause form (see 4.3.7 below).

4.3.7 Before and Behind

The **before** and **behind** keywords are syntactic constructs denoting assignments that insert elements into sequences. They can be completely eliminated without loss of power. **Before** and **behind** appear in several places in the grammar, and are eliminated in similar ways in each place.

An assignment statement of the form

```
new v before X[s1, ... ,sN]
```

is an abbreviation for

```
X := X with (before [s1, ... ,sN] := v)
```

(See [Good 85, section 9.3].)

Alteration clauses of the form

```
S with (before [i] := v)
S with (behind [j] := v)
```

are abbreviations for

```
S[1..i-1] @ [seq: v] @ S[i..size(S)]
S[1..j]   @ [seq: v] @ S[j+1..size(S)]
```

Recall that **S[1..0]** and **S[size(S)+1..size(S)]** both denote empty sequences. Thus, these expansions for **before** and **behind** work properly when inserting before the first element of a sequence, or after the last element of a sequence.

4.3.8 Compact Alteration Clauses

Several forms of alteration clauses are defined in Gypsy as abbreviations for less compact alteration clauses.

Two examples of such compacted forms are

```
X with ([i,j,k] := a)

X with ([i]:= a; [j]:= b)
```

which expand to

```
X with ([i] :=
        X[i] with ([j] :=
                   X[i,j] with ([k] := a)))

(X with ([i]:=a)) with ([j] := b)
```

respectively.

Applying the syntactic transformation of alteration clauses to function calls to the second example, we have:

```
gypsy_alter_element# (gypsy_alter_element# (x, i, a), j, b)
```

4.3.9 EACH Clauses in Alterations

Alternation clauses may contain **EACH** clauses. This allows a succinct description of a large number of regular changes to a structured object. For example, in the context of these declarations

```
type array_index = integer (0..4);
type my_array    = array (array_index) of integer;
function F (n : integer) : integer = pending;
```

and a variable **A** declared to be of type **my_array**, the alteration clause

```
A with (each i:array_index, [i]:=F(i))
```

represents an array in which every element **i** is equal to **F(i)**. This form is an abbreviation for a sequence of alterations to each of the indexed elements. In this case, the expansion would be

```
A with ([0]:=F(0); [1]:=F(1); [2]:=F(2); [3]:=F(3); [4]:=F(4))
```

The bound identifier **i** in the each clause takes on successive values of the specified index type, which must be a bounded simple type, from the smallest to the largest. Since the number of elements modified in the structure must be finite, and the order is specified, we can expand the abbreviated alteration clause into a finite number of simple alterations (without an each clause).

4.3.10 Global Constants

For proof purposes, global constant definitions can be reduced to parameterless functions. The constant declaration

```
const number_of_nodes:integer := 5
```

behaves just as the function declaration

```
function number_of_notes : integer =
   begin
     exit result = 5;

     result := 5;
   end.
```

Note that the value of the constant must be of the declared type.[27]

4.3.11 Unwinding Nested Function Calls in Executable Code

Whenever function calls appear in a statement, the function call is explicitly replaced with a new temporary variable, and the statement is preceded by an explicit assignment of the value of the function call to that new variable. This procedure should be performed from right-to-left within the statement, so that the left-to-right order of parameter evaluation is reflected in the order of the computation of the temporary variables. Repeating this treatment on the generated assignment statements unwinds nested function calls in the proper order. This recursive treatment is terminated when all nested function calls have been removed, and all function calls appear only on the right-hand side of assignment statements.

IF expressions occurring within the right-hand side should be treated as function calls, until unwinding yields an assignment statement with the **IF** expression as the top level of the right-hand side. Then the reduction for **IF** expressions (below) is applied.

Finally, we will be left with assignment statements of the form

- X := Y, or
- X := F(Y, Z, ...).

On the left-hand side, **X** stands for any identifier. On the right-hand side, **Y** and **Z** stand for identifiers, and **F** stands for any function name. Thus, the left-hand side of the assignment is always a simple variable name without any component selectors, and the expression on the right-hand side contains at most one reference to a function name.

If the right-hand side of a reduced assignment statement is of the form

```
X := F(Y, Z, ...)
```

then we add the statement

```
PROVE F_Entry(Y, Z, ...)
```

before the assignment statement, where **F_Entry(Y, Z, ...)** represents the entry specification of **F** applied to the actual parameters in the function call.

4.3.12 IF expressions

An **IF** expression on the right-hand side of an assignment statement can be converted to an **IF** statement with the two branches being assignment statements each with one of the alternative values. Thus,

```
X := if B then C else D fi
```

can be rewritten as

[27] A constant for an abstract type must be given an abstract value of that type, not a concrete value. Hence it is proper to use the **exit** specification above, rather than **cexit**. See chapter 9 for details on data abstraction.

```
if B then X := C
     else X := D
end
```

4.3.13 NEW Statement

The **NEW** statement is a form of assignment that can create a new element of a dynamic object (i.e., a sequence, set, or mapping). It is normalized to an equivalent assignment statement using an alteration clause to describe the new dynamic object. The normalizations are:

```
NEW Y INTO M[X]    {mappings}   ==>   M := M with (into [X] := Y)
NEW X INTO SET S   {sets}       ==>   S := S union (set: X)
NEW X BEFORE S[I]  {sequences}  ==>   S := S with (before [I] := X)
NEW X BEHIND S[I]  {sequences}  ==>   S := S with (behind [I] := X)
```

4.3.14 GIVE Statement

The **GIVE** statement is simply a syntactic abbreviation for a **SEND** statement followed by a **REMOVE** statement. The expansion is illustrated by

```
give S[i] to B;
```

becoming

```
send S[i] to B;
remove S[i]
```

This syntactic form is provided so that Gypsy compilers can easily recognize this form and produce optimized code.

4.3.15 REMOVE Statement

The **REMOVE** statement allows removal of an element from a dynamic structure. This effect is accomplished in normalized form by an explicit assignment of an alteration expression. For example, if **S** is a sequence, then the Gypsy statement

```
remove S[i]
```

becomes

```
S := S with (seqomit [i]).
```

This reduction is also done for removal of elements from sets and mappings using appropriate alteration clauses for the different types of structures.

4.3.16 LEAVE Statement

LEAVE statements are replaced by **SIGNAL *LEAVE**.

4.3.17 LOOP Statement

Each **LOOP** statement is encapsulated in a condition handler for the condition ***LEAVE**. Thus,

```
loop
  <loop body>
end
```

becomes

```
begin

  *loop
    <loop body>
  end
when
  is *Leave;
end
```

4.3.18 Reduction of Statement Lists to PROG2

All statement lists (i.e., sequential compositions of statements) are reduced to nested **`PROG2`** statements. Thus, for example, the composition

```
BEGIN ST1; ST2; ST3 END
```

can be reduced to

```
PROG2 ST1;  prog2 st2; st3 end END
```

Since composition is associative, the **`PROG2`** nesting could just as well be

```
PROG2  prog2 st1; st2 end;  ST3 END
```

4.4 Normalizing Condition Handlers

We briefly describe Gypsy condition handling, and then explain how parallel condition handlers (i.e., a **`BEGIN...WHEN`** statement with arms for several different conditions) can be rewritten as nested condition handlers, each with a single arm.

4.4.1 Abnormal Conditions and Condition Handlers

Conditions are signalled to identify abnormal situations (e.g., when an out-of-bounds index is supplied to the array selector function). When a condition is signalled, all the subsequent Gypsy statements that would normally be executed are traversed and ignored until a condition handler for the signalled condition is found. At that point the condition is "cleared," the appropriate body of the condition handler is executed, and normal execution continues immediately following the **`BEGIN...WHEN...END`** statement. Thus, after execution of

```
begin
  x := 0;
  y := 0;
  signal c;
  x := 1;
when
  is c: y := 1;
  is d: x := 2;
end
```

the variable **`x`** has the value 0, and the variable **`y`** has the value 1. In this particular case, the condition **`d`** can never be signalled, and the statements **`x := 1`** and **`x := 2`** can never be executed.

4.4.2 Reducing Condition Handlers

In the above example of condition handlers, we say that the handlers for **c** and **d** are parallel handlers. Parallel handlers at a particular level correspond to a multiple branch conditional, in which only one of the condition handler bodies can be executed. In particular, if the execution of a handler body signals a condition, that condition will not be trapped by a parallel handler, but by a higher level handler associated with a containing compound statement.

For convenience, we can model parallel condition handlers as a multiple branch conditional, which traps the condition and records it in a variable, and a separate multiple branch conditional, which selects and executes the appropriate handler body based on that variable. The motivation for splitting the condition handler into these pieces arises because the handler body is executed outside the scope of the other parallel handlers. Our execution model is basically stack structured, and this parallel test in one context, together with the conditional execution in another, is awkward to describe.

For example, the parallel condition handlers

```
begin
  S0;
when is C1: S1;
     is C2: S2;
     ...
     is Cn: Sn;
end
```

can be modeled as

```
begin begin ... begin begin
                        cond_num:=0;
                        S0;
                      when is C1: cond_num:=1
                      end
                when is C2: cond_num:=2
                end
            ...
      when is Cn: cond_num:=n
      end
end
case cond_num
  is 1: S1;
  is 2: S2;
  ...
  is n: Sn;
  else assert cond_num = 0;
end
```

Assuming that **cond_num** is an integer variable, then none of the assignments to **cond_num** can signal a condition. Thus, if any of the conditions **Ci** is signalled, it must have been from the compound body **S0**. Any condition not handled by the original parallel handlers simply propagates to an enclosing handler as before. The variable **cond_num** represents a new identifier that is assumed not to occur in the program before normalizing the parallel condition handlers.[28]

Given this expansion mechanism, we proceed to treat condition handlers as if there can only be a handler

[28]Note that since only one abnormal condition can be signalled at a time, only one new variable per routine need be introduced.

for a single condition around a statement.

Chapter 5
A SIMPLE OPERATIONAL MODEL

Here is a simple operational model for Gypsy programs. This is meant to be intuitively satisfying, and to lend credence to the VC generator based on this model.

First a model is presented for the language without procedures, concurrency, buffers, or abnormal conditions. Then the model is extended to handle procedures and abnormal conditions. Buffers and concurrency are dealt with later, in Chapters 7 and 8.

5.1 Basics of the State Model

Gypsy condition handling plays a key role in how statements affect program states. For this reason two notions are defined.

- **state**: a mapping of identifiers to values, and
- **cstate**: an ordered pair, <condition, state>.

A normalized Gypsy program has been essentially reduced to nine internal statement forms. There are six executable statements: three whose purpose is to change the **cstate**, and three whose purpose is to affect flow of control. In addition, three statement forms have no effect on execution at all; they are for proof purposes only.

1. **x := exp**. This assignment statement changes either
   (a) the state binding of the variable **x** to be the result of evaluating **exp**, or
   (b) the condition component of the **cstate**, if (for example) evaluating **exp** results in a **cvalue** whose condition component is not ***NORMAL**.[29]
2. **SIGNAL C**. The **SIGNAL** statement changes the condition component of a **cstate** to **C** if it was ***NORMAL**. Otherwise the **cstate** is unchanged.
3. **BEGIN stmt1 WHEN IS cond-name: stmt2 END**. The statement **stmt1** is executed on the current **cstate**, and the condition of the resulting **cstate** is examined. If the condition is ***NORMAL,** then that **cstate** is the value of the **BEGIN-WHEN** statement. If the condition is not ***NORMAL**, then the condition is compared with the condition **cond-name** handled in the **WHEN** part. If the condition names are the same, then the condition of the **cstate** is changed to ***NORMAL**, and **stmt2** is executed with the new

[29]The condition component can change to **VALUEERROR**, or **SPACEERROR** if the assignment cannot be completed due to violation of type specifications or run-time resource constraints. Recall that **x** stands for an identifier, not an arbitrary name expression. All assignment statements have been normalized as described in Chapter 4.

resulting `cstate` being the value of the `BEGIN-WHEN`.

And the three statements affecting control are:

4. `PROG2 stmt1; stmt2 END`. The `PROG2` form executes two (possibly compound) statements sequentially.
5. `IF b THEN stmt1 ELSE stmt2 END`. The `IF` statement determines which of two embedded statements is executed.[30]
6. `*LOOP stmt END`. The `*LOOP` statement repeats execution of an embedded statement until a condition is signalled.[31]

And the three statements for proof purposes only are:

7. `PROVE`. The `PROVE` statement indicates that a proposition should be proven for the current `cstate`.
8. `ASSUME`. The `ASSUME` statement indicates that a proposition can be assumed to hold for the current `cstate`.
9. `BREAKPATH`. The `BREAKPATH` indicates a cut point when tracing control flow during VC generation.

We discuss expression evaluation (and function invocation) later.

5.2 Basic Procedure Calls

Gypsy procedure call semantics are similar to the procedure call semantics of FORTRAN, Algol 60, and Pascal, but are greatly simplified by several language restrictions:

- Global variables are not allowed.
- Potentially dangerous aliasing among actual parameters is not allowed.[32]
- Gypsy routines that terminate always terminate in a well-defined state, even in the event of abnormal termination (e.g., run-time arithmetic errors such as overflow).
- Parameter passage is uniformly call-by-value-result.[33]

The only shared objects permitted in Gypsy programs are message buffers. The sharing only creates an alias when several concurrent processes have access to the same buffer. For the moment we will discuss only sequential processes and sequential procedure calls.

---

[30]This form requires an `ELSE` branch. We will use an assignment of the form `X := X` as a no-op to model a one-branch `IF` statement.

[31]Recall that the Gypsy `LEAVE` statement is modeled as `SIGNAL *LEAVE`.

[32]Potentially dangerous aliasing refers to the situation in which the values of two or more actual parameters of a routine overlap in such a way that altering one value has the side effect of altering the other. This violates one of our basic assumptions about variables: they don't change spontaneously. A value of a non-buffer variable only changes by way of an explicit assignment (perhaps via a procedure call). Buffer parameters actually represent shared variables, and the semantics of buffers explicitly deny this assumption. However, local buffer histories do obey this property of data variables, and hence potentially dangerous aliasing of buffer parameters in procedure calls is disallowed, just as it is for data parameters.

[33]Call-by-value-result is equivalent to call-by-reference in Gypsy, except for buffer parameters, which are truly shared variables. This is because Gypsy procedure calls always return "result values" for `VAR` parameters, even when the routine exits "abnormally."

5.3 Reducing Procedure Calls to Function Calls and Assignment

We can extend this simple model for statements to include procedure invocation. Procedures can be viewed as assignment statements. Basically, a procedure computes a function from its input parameters to its output parameters, and performs a parallel assignment on exit. The issue of nondeterminism is deferred briefly until section 5.3.3. Concurrency and data abstraction are deferred until chapters 8 and 9. Abnormal termination of functions is discussed in section 5.6.

Consider some simplified examples in the context of the following declarations.

```
procedure P (x:T1; var y:T2) =
  begin
    ...
    y := F(x);
  end;

procedure P2 (x1:T1; var y:T2; var z:T3) =
  begin
    y := F2(x,y);
    z := F3(x,z);
  end;

function F (x:T1) : T2 = pending;

function F2(x:T1; y:T2) : T2 = pending;

function F3(x:T1; y:T2) : T2 = pending;
```

The procedure call statement

```
P(a, v)
```

is equivalent to the assignment statement

```
v := F(a).
```

Similarly, the procedure call

```
P2(a,v,w)
```

is equivalent to

```
v := F2(a,v);
w := F3(a,w).
```

No interference exists between the two assignment statements, because **v** does not appear in the actual parameter list in the Call to **F3**. However, potential interference is still easily handled. The parallel assignment can be formulated by composing the several functions (**F2**, **F3** in the example) into a single composite function returning the several values as a record structure. The individual fields of the record can then be picked out and assigned to the **VAR** parameters simulating a parallel assignment with no interference.

At this point it is clear that the body of a deterministic procedure always computes a function of this sort. We can mechanically construct such a composite function from any procedure definition. Consider the procedure **Q**:

```
procedure Q (a1, ..., aN:T; var v1, ..., vM:T) =
  begin
    ...
  end;
```

We can assume that all parameters are of type **T** without loss of generality. Let ***body*** represent the body of the procedure **Q**. A series of functions **F1** through **FM** is composed according to the following scheme:

```
function Fi (a1, ..., aN:T; x1, ..., xM:T) : T =
   begin
     var v1, ..., vM:T;
     v1 := x1;
     ...
     vM := xM;
     *body*
     result := vi;
   end;
```

Note that the formal **VAR** parameters **v1**...**vM** have been replaced by **const** parameters **x1**...**xM** in the function header. This is necessary because the variables **vi** can appear on the left-hand side of assignment statements in ***body***. (In Gypsy **const** parameters cannot appear on the left-hand side of assignment statements.) We now construct the composite function **FQ** as

```
Big_Record_of_T = record (v1, ..., vM : T);

function FQ (a1, ..., aN:T;
             x1, ..., xM:T) : Big_Record_of_T =
   begin
     result.v1 := F1(a1, ..., aN, x1, ..., xM);
     ...
     result.vi := Fi(a1, ..., aN, x1, ..., xM);
     ...
     result.vM := FM(a1, ..., aN, x1, ..., xM);
   end;
```

and replace the procedure call

```
Q(a1, ..., aN, v1, ..., vM)
```

with

```
temp := FQ(a1, ..., aN, v1, ..., vM);
v1   := temp.v1;
...
vi   := temp.vi;
...
vM   := temp.vM.
```

Thus, in the absence of abnormal termination conditions and concurrency, we have reduced procedure calls to function calls and assignments to variables.

5.3.1 Aliasing Restrictions

Procedure calls are restricted to avoid potentially dangerous aliasing. The parameter passing mechanism is the only way to introduce aliasing in Gypsy. Because there are no mechanisms for direct manipulation of pointers or addresses, aliasing can be defined almost purely syntactically. In Gypsy syntax, a **<VARIABLE NAME EXPRESSION>** is the syntactic class that can appear on the left-hand side of an assignment statement, or as a **VAR** parameter in a procedure call. This class (as it appears in normalized

Gypsy programs) is defined by the following grammar rules:

```
<VARIABLE NAME EXPRESSION> ::= <IDENTIFIER> { <COMPONENT SELECTORS> }

<COMPONENT SELECTOR> ::= . <FIELD NAME>       |
                         ( <INDEX SELECTOR> )
<INDEX SELECTOR> ::= <EXPRESSION>

<FIELD NAME> ::= <IDENTIFIER>
```

In this notation, **{ <COMPONENT SELECTORS> }** denotes that the syntactic form **<COMPONENT SELECTORS>** can be repeated zero or more times. The occurrences of the symbols dot ("**.**"), open parenthesis ("**(**"), and close parenthesis ("**)**") denote literal occurrences of these characters. Note that because **<INDEX SELECTOR>**s are computed expressions, determination of aliasing may depend on state information. Let us consider a **<VARIABLE NAME EXPRESSION>** to be a sequence of constituents as given by these rules, with **<EXPRESSION>** considered to be an atomic constituent. The functions **first** and **nonfirst** are used to decompose the sequence. We can define a predicate, **OVERLAP**, to identify aliasing between two **<VARIABLE NAME EXPRESSION>**s in a particular **state**.

The predicate **OVERLAP(VNE$_1$, VNE$_2$)** holds in in state **S** if and only if

```
  MATCH(first(VNE_1, first(VNE_2)
& if size(VNE_1) > 1 & size(VNE_2) > 1
     then OVERLAP(nonfirst(VNE_1), nonfirst(VNE_2))
     else TRUE fi
```

where **MATCH** is defined as follows:

(a) if either of the first two arguments is the empty sequence, then **TRUE**;

(b) for identifiers, the two identifiers must match literally;

(c) for literal tokens (i.e., "**(**" and "**)**" and "**.**"), the tokens must match literally;

(d) for expressions, the two expressions must evaluate to equal values in the state **S**.

Aliasing is considered *potentially dangerous aliasing* if a **VAR** parameter overlaps with any other parameter. Potentially dangerous aliasing is only possible in procedure calls, since only procedure calls have **VAR** parameters. When translating procedure calls into function calls, we must be sure that the presence of dangerous aliasing is detected. We can do this by inserting a statement of the form

```
if OVERLAP(VNE_1, VNE_2) then signal ALIASERROR end;
```

before the procedure call expansion, where **VNE$_1$** ranges over all **VAR** parameters in the procedure call, and for each **VNE$_1$**, **VNE$_2$** ranges over all other actual parameters. We will also insert

```
Prove (not OVERLAP(VNE_1, VNE_2))
```

into the expanded program. This **PROVE** statement causes the VC generation process to emit VCs stating that no dangerous aliasing occurs.

Note that this analysis for aliasing must be done before normalization, since the function nesting is eliminated as part of normalization, and the component selector operations are viewed as functions.

5.3.2 Checking Value Restrictions

Gypsy allows definition of certain classes of subtypes. Subrange types are based on simple types (scalar types, integers, or rationals) whose value set is restricted to a limited range. For example **`integer [1..10]`** is a subrange type based on type integer. The other sort of subtype is based on size limit restrictions on dynamic structures (sequences, sets, mappings, and buffers).[34]

The static semantic type checking enforced by the Gypsy parser checks to verify that the type of every actual parameter is consistent with the type declared for the corresponding formal parameter. This means that they must have the same *base types*, essentially the same structure composition of the same primitive types, ignoring range and size restrictions.[35] Analysis of dynamic semantics is required to verify that the actual parameter values to be passed will satisfy any range or size restrictions of the formal parameter type. Assertions to verify this can easily be constructed from the type definition associated with each formal parameter, instantiated on the actual parameter expression, and inserted into the routine body as part of the expansion of the procedure call. Since the details of Gypsy type declarations and type checking are not dealt with here, details of this construction are omitted as well.

5.3.3 Nondeterminism

Now consider that Gypsy procedures are not required to be deterministic. They may behave nondeterministically due to observable effects of concurrency, or by revealing concrete representations of abstract data types.[36] Since procedures need not be deterministic, the resulting new value for a **VAR** parameter may not be determined by a single (deterministic) function. Even though each time the procedure is called the new value can be expressed as a function of the input values, it might not be the same function each time. This can be modeled by defining a new function to encapsulate all of the functions that compute the **VAR** parameter at different activations. This encapsulation function takes an additional parameter, which distinguishes each distinct activation of the procedure. So, the new function can choose among the various functions that compute the value of the **VAR** parameter in different activations. The **`activation_id`**, discussed in Section 7.1, can serve as this additional actual parameter, providing a unique actual parameter at every call site.[37]

Specification functions are required to be deterministic, so that the normal rules for substitution and equality will hold in proofs.[38] Using the encapsulation function described above assures that the functions used to model procedure calls are also deterministic. Thus, any statements about the results of procedure invocations that arise in VCs must be well-behaved.

---

[34]Array indexing sets are not considered size restrictions; arrays with different subrange indexing sets are considered different types.

[35]See the Gypsy language manuals for a full discussion of base types.

[36]The procedure may yield different results for actual parameters that are abstractly equal, but not concretely equal. This only appears nondeterministic when viewed though an abstract equality function. See Chapter 9 for details on data abstraction.

[37]The actual VC generator in the GVE performs an equivalent manipulation. It uses a distinct function name to denote the newly computed value of the **VAR** parameter at each call site. Since the VC generator analyzes a procedure based on a finite number of linear code segments (as we see in Chapter 6), only a finite number of unique function names are required.

[38]This is not a language restriction, but a requirement of the implemented proof methods. The soundness of the proofs performed by the GVE depends on this fact. The GVE requires proof that specification functions do not reveal concrete details of abstract data types; specification functions cannot involve concurrency; and they cannot include procedural statements, such as the **AWAIT**. Thus, they must behave deterministically.

5.4 Notation and Preliminary Definitions

In preparation for defining the operational model of Gypsy, we need some basic definitions.

5.4.1 States and Conditions

Definition: A **`state`** is a mapping from identifiers to values.

Definition: A **`cstate`** is an ordered pair, composed of a condition name and a state.

Function: **`CSTATE`** `: condition.name` $\times$ `state` $\rightarrow$ `cstate.`

Definition: A **`cvalue`** is an ordered pair, composed of a condition name and a value.

Function: **`CVALUE`** `: condition.name` $\times$ `value` $\rightarrow$ `cvalue.`

Function: The primitive function **`eval`**: **`identifier`** $\times$ **`state`** $\rightarrow$ **`value`**

maps identifiers to their values with respect to a given state.

We can extend **`eval`** in a natural way to take a **`cstate`**, and return a **`cvalue`**. This extended eval maps the identifier according to the state component of **`cstate`**, and preserves the condition component of **`cstate`** in the resulting **`cvalue`**. We can also later extend **`eval`** to map expressions to values with respect to a given state.

5.4.2 General Notation

We use LISP-style, prefix notation [McCarthy 65]. The notation for function application is **`(G X)`**, which denotes the application of the function **`G`** to the argument **`X`**. We can sometimes use **`X.F`** to denote accessing a field named **`F`** of an n-tuple **`X.`** The same field name can also be used as a function that yields the value of that field, as in **`(F X)`**. Although the name is overloaded, no ambiguity results.

Identifiers and function names are composed of letters, periods (`.`), and pound signs (**`#`**).

Sets can be written by explicitly enumerating its elements within braces, separating the elements by commas. An example is **`{alpha, beta, gamma}`**. Sets can also be defined by describing the set elements with a boolean expression or predicate. For example, **`{i |`** $\exists$ **`j, j is an integer`** $\wedge$ **`i=j*j}`** denotes the set of perfect squares.

Fragments of Gypsy programs are enclosed in double-angle brackets, as ⟨⟨**`x`**⟩⟩. In these Gypsy fragments, words in upper-case will generally denote themselves (Gypsy reserved words), and words in lower-case denote syntax variables or meta-symbols.

The following are some basic definitions:

Definition: **`nil`** denotes the empty set, or the empty list. This is sometimes written explicitly as **`empty.set`** to emphasize representing the empty set.

Definition: **`(cons E L)`** is similar to the LISP **`cons`** function. It extends the list **`L`** by adding the element **`E`** at the beginning.

Definition: **`(rcons L E)`** is "right cons". It extends the list **`L`** by adding the element **`E`** at the end

Definition: **(first L)** returns the first element of the list **L** for non-empty lists.

Definition: **(rest L)** returns all but the first element of the list **L**. **rest** returns **nil** for empty lists.

Definition: **(append L1 L2)** appends lists **L1** and **L2**.

Definition: A **stmt** is a piece of a normalized Gypsy program, corresponding to program text derived from the non-terminal **<STATEMENT>** in the Gypsy grammar.

5.5 Basic Axioms for EXECUTE

The function **EXECUTE** maps a **cstate** and a **stmt** to a new **cstate**. **EXECUTE** is the basic function in our operational model of sequential Gypsy. For the moment we will confine ourselves to the realm of deterministic programs, so **EXECUTE** can be described as a function. We will also restrict our initial discussion to programs that terminate normally. (See section 5.6 for the effect of abnormal termination on the definition of **EVAL**.)

The function **EVAL** maps an expression and a cstate into a value. We can define **EVAL** as follows:

```
W1. (EVAL ⟨⟨identifier⟩⟩ cs) ≡ (lookup identifier cs)

W2. (EVAL ⟨⟨F(x)⟩⟩, cs) ≡
              (lookup RESULT
                (execute (cstate *NORMAL (bind-formal F (eval x cs)))
                         (extended-routine-body F))
```

The function **LOOKUP** takes an **identifier** and a **cstate**, and yields the value of the **identifier** in the state. The function **EXTENDED-ROUTINE-BODY** maps a Gypsy routine name into its normalized, extended body.

Recall that after normalization, function calls are never nested. Therefore, the actual parameters must be simple identifiers. The function **bind-formal** takes a Gypsy function name and a value, and yields a **cstate**.[39] This **cstate** has a single identifier binding, namely the formal parameter name of the Gypsy function named is bound to the second argument of the **bind-formal** call. (We can assume the function takes a single argument without loss of generality, as multiple arguments can always be composed into a simple record structure for parameter passage.) The function **extended-routine-body** is assumed to return the extended routine body derived from a function declaration. When a Gypsy function terminates, the reserved identifier **RESULT** is bound to the value to be returned; therefore, we look up **RESULT** in the final state after executing the routine body.

Function: **Execute** : cstate $\times$ stmt $\rightarrow$ cstate

Here are the axioms defining **Execute**, which formalize the effects of "executing" a Gypsy program.

```
X1. (EXECUTE CS ⟨⟨ASSIGN X exp⟩⟩) ≡
                          (if (= (CONDITION CS) *Normal)
                              then (ALTER CS X (EVAL exp CS))
                              else CS)
```

Recall that the left-hand side of an assignment is always an identifier after program normalization. Thus,

[39] Obviously it also takes an additional, implicit parameter, namely the set of routine declarations.

the symbol **exp** above represents an either an identifier or an unnested function call.

Function: ALTER : cstate × identifier × value → cstate

Definition: For all X, Y : identifiers,

```
(1) (EVAL X (ALTER CS X VAL)) ≡ VAL;
(2) (literal-neq X Y) → (EVAL X (ALTER CS Y VAL)) ≡
    (EVAL X CS).
```

This use of **literal-neq** denotes literal inequality of identifiers, not inequality of the values of the identifiers.

```
X2. (EXECUTE CS ⟨⟨SIGNAL c⟩⟩) ≡ (if (= (CONDITION CS) *NORMAL)
                                     then (cstate c (STATE CS))
                                     else CS)

X3. (EXECUTE CS ⟨⟨BEGIN stmt1 WHEN IS c: stmt2 END⟩⟩) ≡
       (if (= (condition CS) *NORMAL)
           then (if (= (condition cs1) c)
                    then (execute (cstate *NORMAL (state cs1)) stmt2)
                    else cs1)
           else CS)
```

where cs1 = (execute cs stmt1).

```
X4. (EXECUTE CS ⟨⟨PROG2 stmt1; stmt2 END⟩⟩) ≡
                             (execute (execute CS stmt1) stmt2)

X5. (EXECUTE CS ⟨⟨IF b THEN stmt1 ELSE stmt2 END⟩⟩) ≡
                (if (= (CONDITION CS) *Normal)
                    then (if (= (eval b cs) TRUE)
                             then (EXECUTE CS stmt1)
                             else (EXECUTE CS stmt2))
                    else CS)

X6. (EXECUTE CS ⟨⟨*LOOP stmt END⟩⟩) ≡
             (if (= (condition CS) *NORMAL)
                 then (execute (execute CS stmt)
                               ⟨⟨*LOOP stmt END⟩⟩)
                 else CS)
```

Recall that **LEAVE** statements have been converted into **SIGNAL *LEAVE**. Thus, when the body of the loop executes a **LEAVE**, the state condition changes from ***NORMAL** to ***LEAVE**. Also, as part of normalization, the ***LOOP** is encapsulated inside a **WHEN** handler that traps ***LEAVE** and transforms it to ***NORMAL**, so that execution can continue normally after the loop.

```
X7. (EXECUTE CS ⟨⟨PROVE assertion⟩⟩) ≡ CS

X8. (EXECUTE CS ⟨⟨ASSUME assertion⟩⟩) ≡ CS

X9. (EXECUTE CS ⟨⟨BREAKPATH⟩⟩) ≡ CS
```

5.6 Extending EXECUTE to Handle Abnormal Termination

So far our axioms assume that the only way an abnormal condition can arise is from executing a **SIGNAL** statement. Expression evaluation is always assumed to yield a value.

Instantiation of procedure calls can easily be extended to handle condition parameters and case exit specifications. Basically, the procedure body is encapsulated in a **WHEN** handler that traps the formal condition name and then signals the actual condition name (into what was the "calling environment"). Since Gypsy data parameter values are returned even on abnormal termination, the **WHEN** handler must also assign the current values of the (formal) **VAR** parameters to their (actual) variables in the calling environment.

Let us first look at modeling normal expression evaluation in a functional form. First we do this under the assumption that all functions are totally defined. We extend this to include procedure calls within the context of totally defined functions. Then we look at how to extend this model to handle abnormal termination as embodied in Gypsy functions and procedures. Finally, the new axioms for **EVAL** and **EXECUTE** to handle abnormal termination are presented.

5.6.1 Simple Expression Evaluation

Gypsy expressions can be mapped into function calls if all functions are required to be totally defined. The built-in infix operators are defined (in the Gypsy 2.1 report, [Good 85]) to be syntactic abbreviations for calls to predefined functions. For example,

```
i+j*k
```

is an abbreviation for

```
integer#add(i, integer#multiply(j, k))
```

Similarly, accessing data structures, such as arrays and records, is defined in terms of function references with syntactic abbreviation. Thus all Gypsy expressions are really nested function calls. If each function is required to be total, there can be no problem with undefined values creeping into our universe.

5.6.2 Abnormal Termination in Functions

Gypsy functions have no side effects. This discipline is enforced by the fact that functions have no **VAR** parameters, and Gypsy does not allow global variables. A function terminates abnormally if during its evaluation an exception condition is signalled in its body. Rather than returning a value to the call site, the exception condition name is mapped according to the formal/actual condition parameter names and the resulting name is resignalled in the calling environment. (If the exception condition name does not appear in the function's formal condition parameter list, then the standard condition **RoutineError** is signalled in the calling environment.) Signalling in the calling environment aborts all pending function invocations in the current expression evaluation, and control proceeds to the appropriate exception handler (remapping condition names if routine boundaries are crossed).

The first part of modeling abnormal function termination is to return a condition name, and to abort evaluation of the current expression. We do this by extending the value returned by the function to include a condition name. For a function **F** returning a value of type **T1**, **F** is extended to return a value of type **T2**,

```
type T2 = record (Value         : T1;
                  ConditionName : ConditionNames);
```

This is done uniformly for all functions, including predefined ones. Each function body effectively gains a new local variable named **Result.ConditionName**. The **ConditionName** field is initially **Normal**, and is assigned some other condition name when that exception condition would have been signalled. Thus, whenever an exception condition would have been signalled, **Result.ConditionName** is assigned the name of the condition to be signalled, and the remaining portion of the function body is skipped. Consider the function F shown in Figure 5-1. This can be

```
function F (x,y:T) : T unless (cond C1, C2) =
  begin
    result := G(x,y) unless (C1);
    result := H(x,y) unless (C2);
    ...
  end;
```

Figure 5-1: Original Function with Condition Parameters

transformed into the function FCond as shown in Figure 5-2. Note that FCond must check that its actual

```
function FCond (x,y:TC) : TC =
  begin
    var temp:TC;
    if   x.ConditionName ne Normal then
            result.ConditionName := x.ConditionName
    elif y.ConditionName ne Normal then
            result.ConditionName := y.ConditionName
    else
         temp := GCond(x,y);           { MUST HANDLE C1 }
         if temp.ConditionName ne Normal then
            result.ConditionName := temp.ConditionName
         else
            result.value := temp.value;
            temp := HCond(x,y);        { MUST HANDLE C2 }
            if temp.ConditionName ne Normal then
               result.ConditionName := temp.ConditionName
            else
               result.value := temp.value
            end
         end
    end
  end {of FCond}
```

Figure 5-2: Augmented Function without Condition Parameters

parameters are evaluated properly to yield values, as indicated by their condition fields being **Normal**. If the evaluation of one of the actual parameters "signalled" an exception condition during evaluation, then the evaluation of FCond is skipped, and the exception condition is propagated in the result of FCond.

We have added the restriction that actual parameter lists are evaluated in left to right order. This is done precisely so that the IF-tests at the beginning of **FCond** can be performed as illustrated in Figure 5-2. The Gypsy 2.0 report specifies that the order is not defined. Thus, if evaluation of two or more actual parameter expressions terminates abnormally, the condition name propagated is not determined. The motivation for this nondeterministic behavior is to permit the implementors some leeway in evaluating parameter lists in any convenient way. Since expression evaluation can have no side effects other than signalling an exception condition, there is no detectable effect if all the expressions evaluate normally. Nondeterminism complicates our semantics considerably, and we shall endeavor to suppress nondeterministic behavior as much as possible in order to make the semantics more readable. The problems of efficiently compiling the resulting strictly defined language are addressed briefly later. (See also [McHugh 83].)

5.6.3 Extended Axioms for EVAL

The function **EVAL** can be extended to handle abnormal termination in function calls. We first extend **EVAL** to yield a **cvalue**, rather than a **value**. This allows us to return the condition from the final state of a function evaluation, as well as the value of **RESULT**. The new axioms for **EVAL** are:

```
W1'. (EVAL ⟨⟨identifier⟩⟩ CS) ≡
           (if (= (condition CS) *NORMAL)
               then (cvalue *NORMAL (lookup identifier CS))
               else (cvalue (condition CS) dont.care))
W2'. (EVAL ⟨⟨F(x)⟩⟩, CS) ≡
        (if (= (condition CS) *NORMAL)
            then (cvalue (condition final-state)
                         (lookup RESULT final-state))
            else (cvalue (condition CS) dont.care))

        where final-state = (execute (cstate *NORMAL
                                             (bind-formal F (eval x CS)))
                                      (extended-routine-body F)).
```

Observe that the inner call to **EVAL** cannot fail, as it is merely a call to **lookup**. The identifier **dont.care** denotes an arbitrary value. It is used when the value portion of a **cvalue** is not relevant to the computation.

5.6.4 Extended Axiom for EXECUTE

Only the **EXECUTE** axiom for assignment need be extended, since all function calls appear on the right-hand sides of assignment statements in normalized programs. Thus, any abnormal condition that arises from a function call is signalled from an assignment statement. The new axiom is:

```
X1'. (EXECUTE CS ⟨⟨ASSIGN X exp⟩⟩) ≡
        (if (= (condition CS) *Normal)
            then (if (= (condition rhs-cvalue) *Normal)
                     then (ALTER CS X (value rhs-cvalue))
                     else (cstate (condition rhs-cvalue) (state CS)))
            else CS)

        where rhs-cvalue = (EVAL exp (state CS)).
```

For the parallel assignment used to model procedure calls the axiom is slightly different. This reflects the fact that **VAR** parameters in procedure calls change their values as a result of the call *even if the procedure terminates abnormally.* The new assignment axiom for assignment derived from procedure calls is:

```
X1''. (EXECUTE CS ⟨⟨ASSIGN X exp⟩⟩) ≡
           (if (= (condition CS) *Normal)
               then  (cstate (condition rhs-cvalue)
                             (state (ALTER CS X (value rhs-cvalue))))
               else CS)

        where rhs-cvalue = (EVAL exp (state CS)).
```

Chapter 6
A VERIFICATION CONDITION GENERATOR

6.1 VC Generation

Verification conditions are propositions that state the consistency between a program and a specification. Proving the VCs is sufficient to prove that the program satisfies the specifications. Just as we started with a simple model of execution and expanded it to handle more complex aspects of the language, we will start with a basic model of VC generation and expand that model.

We can derive a description of VC generation from the definition of **EXECUTE**. VC generation basically involves symbolic execution of each possible path in the program. Since there are potentially an unbounded number of possible paths through a routine (because of **LOOP** statements and recursion), we cannot truly symbolically execute each possible path. Therefore, we divide VC generation into two steps:

- computing the finite set of finite linear path segments from which all other control paths through the routine can be composed, and
- symbolic execution of those path segments.

This separates the process of control path analysis from the process of symbolic execution, which allows the control path analysis to be done on a purely syntactic basis, leaving the interpretation of dynamic semantics and specifications to symbolic execution. The actual symbolic execution is really quite simple, once the appropriate set of path segments has been computed.

Proving properties of loop paths corresponding to an arbitrary number of iterations though the loop body requires an inductive proof. For example, let the regular expression $\mathbf{A\ B^{*}\ C}$ represent the set of paths traversing A, followed by any number of traversals of B, followed by a traversal of C. We can prove properties of this infinite set of paths by using induction on the form of each path, using **P(AC)** as the base case, and

$$\mathbf{P(A\ B^{n}\ C)} \rightarrow \mathbf{P(A\ B^{n+1}\ C)}$$

as the inductive step. In general the symbols **A**, **B**, and **C** denote sets of paths, since there can be multiple paths through the loop body. **ASSERT** statements are used in the program to indicate the loop body is to be decomposed into these segments, and to specify the property **P** to be proven. Thus, all iteration paths through loop bodies must be broken by an **ASSERT** statement. (This use of the **ASSERT** statement becomes more explicit in normalized Gypsy programs by the presence of the **BREAKPATH** statement form.)

This set of path segments, which "covers" all possible paths through the program is called the *covering* *path set*.[40] We use **ASSERT** statements to break paths into path segments. Each path segment starts with an **ASSERT** statement, and continues until it encounters another **ASSERT** statement. A **SIGNAL** statement causes subsequent statements to be ignored until an appropriate **WHEN** handler is encountered. Basically, wherever **EXECUTE** is defined to choose between two possible execution paths, our path tracing function traces both paths (yielding a set of possible paths, where **EXECUTE** traced and executed a single path). And the state change operators (assignment, signal, and when-handler) are interpreted. **ASSERT** statements are predicates about the program state.

The VCs are propositions in first order logic that state that for each path the initial assertion evaluated in the null state[41] implies the validity of the final assertion evaluated in the state resulting from applying the sequential composition of all of the state change operators along the path.

6.2 Preliminary Definitions for Manipulating Paths

Here are some additional definitions that will be used to define the functions that construct the set of linear path segments derived from a Gypsy routine. Hereafter, the term "path" will be used to mean a "linear path segment".

Definition: **(pair x y)** denotes an ordered pair, sometimes written **<x, y>**.

Definition: **(pair.union (pair x_1 y_1) (pair x_2 y_2) ...)** ≡
(pair (union x_1 x_2 ...) (union y_1 y_2 ...)).

Definition: A **StmtList** is a piece of a normalized Gypsy program, corresponding to program text derived from the non-terminal **<STATEMENT LIST>** in the Gypsy grammar.

Definition: A **path** is a pair **<Condition, StmtList>**. A path object is constructed by the function **path**.

Function: **Path** : condition × stmtlist → path.

Path is also used with a single statement as its second argument. This simplifies our notation, and does not introduce any ambiguity.

Function: **Condition** : path → condition

Function: **StmtList** : path → stmtlist

Definition: If **P** is the path **(Path c s)**, then **(Condition P) = c**, and **(StmtList P) = s**.

Definition: The **empty path** is the path **(path *normal nil)**, and is sometimes denoted by **empty.path**.

Definition: **(cons.path P E)** is defined to be **(path (condition P) (cons (StmtList P) E))**, for path P and path

[40] This is called the set of Floyd paths by Boyer and Moore in their description of a VC Generator for FORTRAN. [Boyer&Moore 81b].

[41] In a null state, all identifiers evaluate to themselves.

element E.

Definition: **(cons.paths.in.set S E)** is defined to be **{(cons.path P E) | P ∈ S}** for **S** a set of paths, and **E** a statement.

Definition: **(rcons.path P E)** is defined to be **(path (condition P) (rcons (StmtList P) E))** for **P** a path, and **E** a statement.

Definition: **(rcons.paths S E)** is defined to be **{(rcons.path P E) | P ∈ S}**, for **S** a set of paths, and **E** a statement.

Definition: **(append.paths P1 P2)** is defined to be

```
(path (condition P2)
      (append (StmtList P1) (StmtList P2)))
```

for paths **P1**, and **P2**.

Definition: **(append.paths.in.set S1 S2)** is defined to be

```
{(append.paths P1 P2) |   P1 ∈ S1
                        ∧ P2 ∈ S2}
```

for sets of paths **S1** and **S2**.

Function: Normal.Paths: set.of.paths → set.of.paths.

Definition: **(Normal.Paths S) ≡ {P | P ∈ S ∧ (condition P) = *normal}.**

Function: Abnormal.Paths: set.of.paths → set.of.paths.

Definition: **(Abnormal.Paths S) ≡**
{P | P ∈ S ∧ (Condition P) ≠ *normal}.

Function: Signal.Paths : set.of.paths × condition-name → set.of.paths.

Definition: **(Signal.Paths S C) ≡ { P | P ∈ S ∧ (condition P) = C}.**

Function: NonSignal.Paths : set.of.paths × condition-name → set.of.paths.

Definition: **(NonSignal.Paths S C) ≡**
{ P | P ∈ S and (condition P) ≠ C}.

6.3 Tracing Paths: The Functions CPS and EP

The function **EXECUTE** defines all of the control paths potentially followed during execution of a Gypsy routine. We define a *linear path segment* to be a potential control path that begins and ends with a **BREAKPATH** statement.[42] Routines with loops or recursion can describe potentially infinite control paths. However, there can only be a finite number of distinct linear path segments for any Gypsy routine. This

[42]Boyer and Moore [Boyer&Moore 81b] call this a Floyd path.

follows because the text of the Gypsy routine is finite, and potentially infinite control paths can only be composed through recursion (in which case the paths are broken by the entry and exit specifications of the routine) or iteration (and we assume that each **LOOP** statement is properly sprinkled with **ASSERT** statements, which yield **BREAKPATH** statements to break each potential path through the body of the loop). Thus, all possible execution paths for a routine are composed of linear path segments. The set composed of these linear path segments is called the ***covering path set*** of the routine.

We want to define a function **CPS** that computes the covering path set for a Gypsy routine. These paths cover all possible execution paths through the routine, so that it is sufficient for proof purposes to deal with these finite path segments, even to prove properties of possibly infinite execution paths.

The body of a Gypsy routine is a single (compound) statement. Thus, we want to be able to compute the linear path segments through a single (possibly compound) statement. We further reduce the problem to extending a set of paths by a single statement. We call this "extend path" function **EP**. (**EP** computes the control paths on a purely syntactic basis, and so can yield paths that are never, in fact, traversed during execution.)

Function: CPS : `stmt` $\rightarrow$ `set.of.paths,` the covering path set for the statement.

The path segments each begin and end where an assertion (**ENTRY**, **EXIT**, or **ASSERT** statement) appeared in the program. These points where paths must be broken are indicated explicitly in the normalized Gypsy program by the **BREAKPATH** statement. Thus, as we build the covering path set, we keep two partial results:

- **finished paths**, which begin at an assertion and end with a **BREAKPATH**, and
- **partial paths**, which begin with an assertion, but do not end with a **BREAKPATH**.

The function **EP** takes three arguments, a set of partial paths, a set of finished paths, and a Gypsy statement. **EP** yields a new set of partial paths, and a new set of finished paths. When computing the covering path set, we want to extend each partial path as we encounter additional statements. When a partial path is extended with a **BREAKPATH** statement, it is added to the set of finished paths. Path segments in the set of finished paths do not get extended as additional statements are processed.

Function: EP : `partial.paths` $\times$ `finished.paths` $\times$ `stmt`
$\rightarrow$ `partial.paths` $\times$ `finished.paths,`

> extends all of the normal paths in **`partial.paths`** by adding **`stmt`** at the end, yielding the resulting new partial paths, and possibly some additional finished paths.

EP extends each of the normal paths in **`partial.paths`** by one additional statement, and yields two new partial results. Path segments in **`finished.paths`** argument are not affected by **EP**, and appear in the **`finished.paths`** of the result. If the additional statement is an **ASSERT**, then any partial path extended by the **ASSERT** becomes finished, and is included in the set of finished paths yielded by **EP**.[43]

For ease of notation, the first two arguments of **EP** are combined into an ordered pair. Thus, **EP** takes an ordered pair of sets of paths as its first argument, and yields an ordered pair of sets of paths as its result. The two components of the result of **EP** can be extracted by the functions **Partial.Paths** and **Finished.Paths**.

---

[43]Some partial paths may not be affected by the **ASSERT,** if their associated condition is abnormal. Exactly how this works is discussed shortly.

Definition: **`If p is the pair (pair x y), (Pfirst p) = x and (Psecond p) = y.`**

Definition: **`If p is the pair (pair x y), (Partial.Paths p) = (Pfirst p) and (Finished.Paths p) = (Psecond p).`**

Definition: **`(CPS Stmt)`** ≡ **`(Finished.Paths (EP (pair {empty.path} empty.set) stmt)).`**

6.4 The Axioms for EP

We basically derive the axioms for **`EP`** from the axioms for **`EXECUTE`**. Where **`EXECUTE`** uses state information to choose between possible control paths, **`EP`** traces both paths, adding as an assumption the predicate (or its negation) that **`EXECUTE`** uses to choose the proper control path.

The function **`EP`** is defined by the following equations.

Since **`EP`** extends each normal path in **`p`**, the **`partial.paths`** component of its result is the union of the extended normal paths of **`p`**, and the abnormal paths of **`p`** (which are not affected, and remain partial paths). Since extending a partial path by an assignment statement cannot result in a finished path, the **`finished.paths`** component of the result is simply the set of finished paths that was passed as an argument, namely **`f`**. Tracing abnormal exit paths from a function call in **`exp`** will be discussed in section 6.5.

```
E1.  (EP (pair p f) ⟨⟨X := exp⟩⟩) ≡
         (pair (union (rcons.paths (Normal.Paths p) ⟨⟨X := exp⟩⟩)
                      (Abnormal.Paths p))
               f)
```

The **`SIGNAL`** statement changes the condition of each normal partial path, leaving abnormal partial paths unaffected.

```
E2.  (EP (pair p f) ⟨⟨SIGNAL C⟩⟩) ≡
         (pair (union (Abnormal.Paths p)
                      (Change.Condition (rcons.paths NP ⟨⟨SIGNAL C⟩⟩)
                                        C))
               f),
        where NP = (Normal.Paths p).
```

Function: Change.Condition : `set.of.paths × condition-name → set.of.paths.`

Definition:
```
(Change.Condition S C) ≡
     { P1 |    (∃ P ∈ S, (StmtList P) = (StmtList P1))
             ∧ (condition P1) = C}.
```

The **`BEGIN`** causes the normal paths of **`p`** to be extended by **`stmt1`**. Any of the resulting partial paths that are signalling condition **`c`** are "caught" by the **`WHEN`**, and extended by **`stmt2`**. Any of the new partial paths generated by adding **`stmt1`** that are normal, or signal a condition other than **`c`** are left as partial paths in the result.

```
E3.  (EP (pair p f) ⟨⟨BEGIN stmt1 WHEN IS c: stmt2 END⟩⟩) ≡
         (pair (union (Abnormal.Paths p)
                      (Paths.Not.Signalling (Partial.Paths A) c)
                      (Partial.Paths B))
               (union (Finished.paths A)
                      (Finished.paths B)))
```
[44]

where A = (EP (pair p f) ⟨⟨**stmt1**⟩⟩),
and B = (EP (pair (Paths.Signalling (Partial.Paths A) c) nil) ⟨⟨**stmt2**⟩⟩).

Function: Paths.Signalling : set.of.paths $\times$ condition $\rightarrow$ set.of.paths.

Definition:
```
(Paths.Signalling S name) ≡
     { P1 |    (∃ P ∈ S, (StmtList P) = (StmtList P1))
             ∧ (condition P)  = name
             ∧ (condition P1) = *NORMAL}.
```

Function: Paths.Not.Signalling : set.of.paths $\times$ condition $\rightarrow$ set.of.paths.

Definition:
```
(Paths.Not.Signalling S name) ≡
            {path | path ∈ S ∧ (condition path) ≠ name}.
```

PROG2 is simply the statement composition operator.

```
E4.  (EP (pair p f) ⟨⟨PROG2 stmt1; stmt2 END⟩⟩) ≡
         (EP (EP (pair p f) ⟨⟨stmt1⟩⟩) ⟨⟨stmt2⟩⟩).
```

In **E5** the evaluation of **bool** must terminate normally, because in a normalized program **bool** must be a simple variable. Any abnormal paths originating as part of the evaluation of the expression from the original **IF** statement are traced from one of the assignment statements generated during normalization of expressions containing function calls.

```
E5.  (EP (pair p f) ⟨⟨IF bool THEN stmt1 ELSE stmt2 END⟩⟩) ≡
        (pair.union (EP (pair (rcons.paths NP ⟨⟨ASSUME bool⟩⟩) f)
                        stmt1)
                    (EP (pair (rcons.paths NP ⟨⟨ASSUME NOT bool⟩⟩) f)
                        stmt2)
                    (pair (Abnormal.Paths p) f)),
```
where NP = (Normal.Paths p).

The path tracing extends paths along both branches of the **IF** statement, placing the appropriate assumption in each path. This can generate paths that seem possible syntactically, but are not possible program executions. For example the fragment

[44] It is not necessary to include f in the second union. It is subsumed by the term ⟨⟨**(Finished.paths A)**⟩⟩, because

(A = (EP (pair p f) <<stmt>>).

Thus set ⟨⟨**(Finished.paths A)**⟩⟩ includes ⟨⟨**f**⟩⟩ as a subset, since each equation defining **EP** preserves the **f** component of its first argument.

```
x := 0;
if x=0 then if x=1 then Stmt1;
                    else Stmt2;
                    end
       else Stmt3;
       end;
```

generates paths through **Stmt1**, even though the two **IF** tests are mutually exclusive, and through **Stmt3**, even though the **IF** test **x=0** evaluates to **FALSE** during symbolic evaluation of the path. In both cases the VCs generated during symbolic execution have **FALSE** as a hypothesis, and hence immediately reduce to **TRUE**.

Recall that in normalized Gypsy programs, **LEAVE** statements have been transformed into **SIGNAL *LEAVE**, and ***LOOP** statements have been encapsulated in when handlers to handle the ***LEAVE** condition.

```
E6.  (EP (pair p f) 〈〈*LOOP stmt END〉〉) ≡
         (EP (EP (pair p f) stmt) 〈〈*LOOP stmt END〉〉).
```

Equation **E6** is clearly circular. There is no ground case for the recursion. **E6** is the most intuitively clear and correct formulation of the paths through the loop. It produces the unbounded set of syntactically possible paths through the loop body, including paths of unbounded length. We only need the component path segments sufficient to construct these possibly infinite paths.

Other equivalent closed-form expressions can yield those component path segments. Let us construct one such closed form. All paths through the body of the loop are broken by **BREAKPATH** statements. Hence, all new partial paths resulting from **(EP (pair p f) stmt)** must begin with an **ASSUME** statement in **stmt**. So, the first recursive reference to **EP** extends these paths that start in **stmt** through the **stmt** as the "second iteration" of the loop. All of the normal partial paths from the previous "iteration" become finished this time around (since all paths through the loop are broken by a **BREAKPATH**) -- or they become abnormal paths.[45] Thus, the normal partial paths resulting from this reference to **EP** are exactly the same as the normal partial paths that resulted from the first call.[46] (I.e., the resulting normal partial paths are completely determined by **stmt**, regardless of the pair **<p,f>** from which we start.) In fact, the same argument applies to the abnormal partial paths, with the exception that **(Abnormal.Paths p)** are maintained throughout the recursion. Abnormal paths correspond to paths that leave the loop (either due to a **LEAVE** statement or some other signal), and normal paths correspond to a continuation of the loop. Thus, we arrive at the closed form rule:

```
E6'.  (EP (pair p f) 〈〈*LOOP stmt END〉〉) ≡
        (pair (union (Abnormal.Paths p)
                     (Abnormal.Paths (Partial.Paths A))
                     (Abnormal.Paths (Partial.Paths B)))
              (Complete.Paths B)),

        where A = (EP (pair (Normal.Paths p) f) stmt),
        and   B = (EP A stmt).
```

[45] The **BREAKPATH** and **ASSUME** must occur because each **LOOP** is required to have a proof-time **ASSERT** statement somewhere along each execution path through its body, and the normalized form of the **ASSERT** yields the **BREAKPATH; ASSUME ...** pair.

[46] In fact, this allows us to detect paths through the loop that are not broken by **ASSERT** statements. If the set of normal partial paths increases in size at the first level recursive call, then it must be that some partial path from the initial call was extended. But that means it was not terminated by an **ASSERT**/**BREAKPATH** statement.

Note that **EP** avoids producing duplicate paths through the loop body by using sets of path segments and set union to remove duplicates that otherwise might be introduced during the second traversal of the loop.

The **PROVE** and **ASSUME** statements simply extend a partial path. They will be interpreted later as indicating that a formula is to be proven (i.e., a VC is to be generated), or that an assumption about the state should be made (i.e., a hypothesis should be added to all subsequent VCs generated along this path.

```
E7.  (EP (pair p f) ⟨⟨PROVE q⟩⟩) ≡
         (pair (union (rcons.paths (Normal.Paths p) ⟨⟨PROVE q⟩⟩)
                      (Abnormal.Paths p))
               f)

E8.  (EP (pair p f) ⟨⟨ASSUME q⟩⟩) ≡
           (pair (union (rcons.paths (Normal.Paths p) ⟨⟨ASSUME q⟩⟩)
                        (abnormal.paths p))
                 f)
```

The **BREAKPATH** statement causes all normal partial paths to become finished paths. **BREAKPATH** only appears where an **ASSERT** appeared in the unnormalized program, and such occurrences are always immediately preceded by a **PROVE** statement. Thus, we know that all finished paths end in a **PROVE** statement that corresponds to an **ASSERT** in the original program.

```
E9.  (EP (pair p f) ⟨⟨BREAKPATH⟩⟩) ≡
         (pair (Abnormal.Paths p)
               (union (Normal.Paths p)
                      f))
```

6.5 Effects of Abnormal Function Termination on EP

In normalized programs all function calls appear on the right-hand side of assignment statements. Hence, only axiom **E1**, the axiom for assignment, needs change. A few new functions to describe the abnormal exits possible from a Gypsy function are required.

Function: Num.Cond.Params: `function.name` $\rightarrow$ `integer.`

This function returns the number of formal condition parameters that the named function accepts.

Function: ExitCase: `function.name` $\times$ `integer` $\rightarrow$ `Boolean.Expression.`

This function returns the exit assertion associated with the N^{th} condition parameter to the function. It returns **TRUE** if there is no exit assertion for a particular condition parameter.

Function: ExitSpec: `function.name` $\times$ `special.cond.name` $\rightarrow$ `Boolean.Expression.`

This function returns the exit assertion of the named function associated with a predefined Gypsy condition name. The predefined Gypsy conditions are **ROUTINEERROR**, **SPACEERROR**, and **NORMAL**. It returns **TRUE** if there is no exit assertion for a particular predefined condition.

Handling abnormal exit paths from function calls adds a number of extra abnormal paths to the partial paths result of **EP**. There are always at least two abnormal paths, corresponding to the predefined conditions **ROUTINEERROR**, and **SPACEERROR**, plus **n** more, where **n** is the number of formal condition parameters accepted by the Gypsy function being invoked.

Consider the assignment statement

```
X := F(X)
```

or

```
X := F(X) unless (C1, C2)
```

in the case of a function call with actual condition parameters. The function **F** can be a predefined Gypsy function or a user-defined function. For each normal partial path in the call to **EP**, for **n** from 1 to **Num.Cond.Params(F)**, the path is extended by ⟨⟨**ASSUME (exitcase F n) on (X); SIGNAL** $\mathbf{c_n}$⟩⟩, where **(exitcase F n) on (X)** denotes the appropriate exit specification for **F** instantiated on the actual parameter list of the function call, and $\mathbf{c_n}$ is the actual condition parameter corresponding to the n^{th} formal condition parameter. Missing actual condition parameters default to **ROUTINEERROR** or **SPACEERROR**, as described in the Gypsy manual [Good 85, section 8.5.3].[47]

6.6 Whatever Became of Procedure Calls?

Procedure calls are reduced to function calls plus assignment when programs are normalized. This simplifies path tracing, and makes assignment the only statement form that alters variable bindings. When we symbolically evaluate the assignment statement, a function reference from the right-hand side is simply recorded as part of the new binding for the variable named on the left-hand side. This is fine for user-defined functions and predefined Gypsy functions, as these have definitions available when proving the VCs, and the function definition can be expanded as part of the proof. But the function definitions corresponding to the reduced procedure calls are not available during the proof. For these functions, an explicit **ASSUME** statement asserting the appropriately instantiated exit specification of the procedure is inserted immediately after the assignment.

In the Gypsy execution model, **EVAL** handled function calls on the right-hand side of an assignment statement by invoking **EXECUTE** on the extended body of the function. Every execution path through that body starts at the **ENTRY** specification and finishes at the **EXIT** specification. Applying the principle of abstraction, we can suppress the computational details of the routine body, and simply assume that the **EXIT** specification supplies sufficient detail about the result of the function invocation to complete the proofs. So, for proof purposes, the function call can be reduced to

1. proving the routine's **ENTRY** specification (instantiated on the actual parameters),
2. assigning a new symbol to the left-hand side of the assignment,
3. assuming the routine's **EXIT** specification (instantiated on the actual parameters, with the new symbol used above taking the place of **RESULT** in the **EXIT**).

This is the actual form that function calls and procedure calls should take in the expanded routine body, which introduces procedural abstraction into our VCs.

Thus, for the procedure **P** declared as

```
procedure P (var x:integer) =
  begin
    entry x in [0..100];
    exit x = x' + 1;
    x := x+1;
  end;
```

the procedure call

[47]Formal **cond** parameters are divided into 5 groups: **VALUE**, **ALIAS**, **COND**, **SPACE**, and **ELSE**. Missing actual condition parameters in the **SPACE** group default to **SPACEERROR**; all other missing condition parameters default to **ROUTINEERROR**.

```
P(N);
```

would expand into

```
N := P#1(N);
```

in the expanded body. Adding the entry and exit properties yields

```
PROVE N in [0..100];
Temp := P#1(N);
ASSUME Temp = N + 1;
N := Temp;
```

P#1 is a new, previously undefined, function symbol, and **Temp** is a previously unused variable name. This expansion allows the proof to use the relation between the input and output values of the **VAR** parameter **X** given in the exit specification of **P**.

Because procedures need not be deterministic, two distinct calls to the same procedure must not yield assignment statements that behave identically during symbolic execution. That is, the symbolic evaluation of the assignment **Temp := P#1(N)** must behave differently each time it is symbolically executed, yielding a different symbolic value for **Temp** each time. We accomplish this by presuming that the generated function symbols (such as **P#1**) can be distinguished from function symbols that originally appeared in the program. Then, each time an assignment statement with one of these distinguished symbols on the right-hand side is executed symbolically, the distinguished symbol is replaced with another arbitrary new symbol. Thus, the only information relating the input and output values of the **VAR** parameters comes from the exit specification of **P**, and there is no deterministic relation between the results of two invocations of **P** (other than what is stated in the exit specification).

6.7 Generating VCs from Paths

To generate the actual VCs from the paths is a simple matter of symbolically executing the path, emitting one VC each time a **PROVE** statement is traversed.

Function: SymEval: Symbolic.Expr × State → Symbolic.Expr.

Definition: (SymEval ⟨⟨x⟩⟩ S) ≡ (lookup x S),

where x denotes the name of a Gypsy variable.

Definition: (SymEval ⟨⟨F(x)⟩⟩ S) ≡ (subst (SymEval ⟨⟨x⟩⟩ S) ⟨⟨x⟩⟩ ⟨⟨F(x)⟩⟩)

Function: subst: Symbolic.Expr × Variable.Name × Symbolic.Expr → Symbolic.Expr.

The function **subst** substitutes its first argument for all occurrences of its second argument within its third argument. For example,

(subst ⟨⟨X⟩⟩ ⟨⟨Y⟩⟩ ⟨⟨F(Y)⟩⟩) would yield **⟨⟨F(X)⟩⟩.**

Function: Generate.VCs: set.of.paths → set.of.vcs

Definition: (Generate.VCs set.of.paths) ≡
{vc | ∃ p ∈ set.of.paths, vc ∈ (generate.path.vcs p)}

Function: generate.path.vcs: `path → set.of.vcs`

Definition: generate.path.vcs path ≡
`(generate.more.vcs empty.set ⟨⟨TRUE⟩⟩ empty.state path)`

Function: generate.more.vcs: `set.of.vcs × boolean.expr × state × path → set.of.vcs`

Generate.more.vcs is defined by the following equations:

```
G0. (generate.more.vcs vcs context state nil) ≡ vcs

G1. (generate.more.vcs vcs state (cons ⟨⟨X := Y⟩⟩ path)) ≡
            (generate.more.vcs vcs
                                   (alter state
                                          ⟨⟨X⟩⟩
                                          (SymEval ⟨⟨Y⟩⟩ state))
                                   path)

G2. (generate.more.vcs vcs context state (cons ⟨⟨ASSUME q⟩⟩ path)) ≡
        (generate.more.vcs vcs ⟨⟨qval AND context⟩⟩ state path),
        where qval = (SymEval ⟨⟨q⟩⟩ state).

G3. (generate.more.vcs vcs context state (cons ⟨⟨PROVE q⟩⟩ path)) ≡
         (generate.more.vcs (union { ⟨⟨context → qval⟩⟩ } vcs)
                                context state path),
        where qval = (SymEval ⟨⟨q⟩⟩ state).
```

6.8 Proof of termination

6.8.1 When It Is Required

It may be desirable to prove termination of user routines to avoid certain unpleasantness at execution time. However, "termination" proofs are required for all definitions of functions that appear in program specifications. This is so because functions used in specifications and proofs must be well-defined, and functions whose recursive definitions do not terminate for some values do not yield well-defined results for those values. Since the proofs use the function defining form contained in the routine's **`EXIT`** specification, that definition must be shown to yield a well-defined function. The executable body of the routine can also be analyzed with regard to proving termination of the executable program, but this is not required for the soundness of the program proof. The following new forms of assertions could easily be added to facilitate generation of the required additional VCs.

6.8.2 Methods (measure functions, count assertions)

Routines can fail to terminate by entering a non-terminating **`LOOP`** statement, or by entering a series of non-terminating routine calls. Our proof methods must detect either sort of failing.

6.8.3 Termination of Loops

Each assertion inside a **LOOP** statement must be extended by a "count" clause. The syntax for **ASSERT** given in [Good 85] is extended from

```
<assert specification> ::= ASSERT <specification expression>
```

to

```
<assert specification> ::= ASSERT <specification expression>
                                  [<count clause>]
<count clause> ::= <non-negative integer expression>
<non-negative integer expression> ::= <expression>
```

where **<non-negative integer expression>** is constrained to be an **<expression>** that yields a non-negative integer value. When a path starts and ends within a loop, both the beginning and ending assertions must have count clauses. If the final assertion of the path has a count clause **exp2** then the VCs include additional conclusion **exp2 > -1** to be proven. If the initial assertion of the path has a count clause(**exp1**), then the additional assumption **exp1 > -1** is made at the beginning of the path. Finally, if both the beginning and ending assertions have count clauses, then the additional conclusion **exp1 < exp2** must be proven. To assure that this decreasing chain cannot be infinite, we must be assured that the initial count is non-negative. Thus, paths entering a loop can have the form

```
assert P;
S1;
assert Q count exp2;
```

and paths within a loop can have the form

```
assert P count exp1;
S1;
assert Q count exp2;
```

and paths leaving a loop can have the form

```
assert P count exp1;
S1;
signal *Leave;
...
assert Q;
```

The count clauses and the resulting VCs are sufficient to establish loop termination, because the count clauses determine a strictly decreasing sequence of non-negative integers and any such sequence must be of finite length.

6.8.4 Termination of Recursive Calls

Since there are only a finite number of routine declarations in a Gypsy program, a non-terminating (i.e., infinite) sequence of routine calls must contain some routine an unbounded number of times. We prevent non-terminating recursion by imposing two requirements. First, all directly recursive routines must have an additional external specification, called a **measure** specification. The measure specification is used similarly to the count clause of a loop assertion, to assure that any sequence of recursive routine calls is finite. The syntax for **<abstract operational specification>** is extended to include an optional **<measure specification>**, where

```
<measure specification> ::=
                      measure <non-negative integer expression>
```

As with all other **`<external operational specifications>`**, only global constants and data parameters can be used as variable identifiers within the **`measure`** specification.

The **`measure`** specification forms an additional conjunct of the entry specification of the routine, much in the manner of the **`count`** clause in loop assertions. At each call site, it must be proven that the **`measure`** specification instantiated on the routine data parameters of that call yields a non-negative integer value. Thus, the entry specification assumed at routine entry may be augmented by the additional assumption that the measure specification instantiated on the routine's formal parameters yields a non-negative integer value.

Further, for recursive calls (i.e., when a routine directly calls itself) it must be shown that the value of the **`measure`** specification instantiated on the parameters of the recursive call is strictly less than the value of the **`measure`** specification instantiated with the routine's formal parameters. Thus, the values of the **`measure`** specification instantiated on the actual parameter values of a sequence of recursive routine calls must yield a non-negative sequence of integer values. Since all such sequences must be finite, the sequence of recursive routine calls must be finite.

6.8.5 Mutual Recursion

The argument of the preceding section must apply to all sequences of recursive routine calls, even if calls to other routines may be interspersed in the sequence. The analysis required to uncover all possible routine calling sequences that potentially contain indirect recursion is potentially exponential in the number of routines in the program. This cost is deemed to be more expensive than the benefit of permitting mutually recursive routines. Therefore, the methods for proving termination of sequential Gypsy programs assume absence of mutual recursion.[48]

---

[48]The GVE is being extended to detect mutual recursion and to issue a warning that termination proofs will not be valid.

Chapter 7
BUFFERS

7.1 What Is an Activation_Id?

A Gypsy program is a collection of program units -- types, functions, procedures, and lemmas. Of these, only the "routines" -- functions and procedures -- are executable. During execution, the executable definitions of various routines are invoked, perform the computation described by the routine body, possibly change the program state, and exit. Such a sequence of actions is called an *activation* of the routine. Each time the routine is invoked, it yields a separate and distinct activation of the routine. The lifetime of a routine activation is that portion of the execution corresponding to computations that are performed between the invocation of this activation of the routine and the exit from this activation. An activation is said to be *alive* during its lifetime, and *dead* during other portions of program execution. Note that during the lifetime of an activation there can be other live activations of the same routine, corresponding to recursive calls beneath this activation or calls to this routine from a parallel process.

An activation_id is a Gypsy object that uniquely identifies a particular activation of a particular routine. The activation_id is used to identify portions of buffer histories representing buffer operations during the lifetime of a routine activation.

If an activation of routine P with activation_id P#1 invokes routine Q (creating an activation of routine Q with activation_id Q#1) during execution, then P#1 is the *parent* of Q#1, and Q#1 is a *child* of P#1. The *ancestors* of an activation are the set formed by the transitive closure of the parent relation. The *descendants* of an activation are the set formed by the transitive closure of the child relation.

7.2 A Model of Activation_Ids

We can model activation_ids as a path identifying a routine activation by listing the routine's ancestors' activation_ids together with a unique "invocation count" from its immediate parent.

For example, activation_ids could be defined as an abstract type.

```
type Activation_id < ... > =
  begin
    ID : sequence of integer;
  end;
```

The activation_id of the top level procedure invoked in at run time might be **`(seq: 0)`**. (The Gypsy notation **`(seq: 0)`** denotes a sequence of integers containing the single element 0.) Recall that each Gypsy routine receives an implicit data parameter of type activation_id under the formal parameter name of **MYID** [Good 85, section 10.7.1]. Thus, within procedure **MAIN** the parameter **MYID** would be bound to the value **(SEQ: 1)**. Each time **MAIN** invokes a routine, it will construct a new activation_id by using the predefined function **New_Activation_Id**. **New_Activation_Id** takes two parameters, an activation_id (i.e., a list of integers), and an integer. It returns a new activation_id, which is formed by attaching the integer parameter to the end of the activation_id parameter. Thus, **New_Activation_Id** might be defined as

```
function New_Activation_Id
        (ID: Activation_id; Ext : integer) : Activation_id =
  begin
    result = ID <: Ext;
  end
```

The Gypsy operator **"<:"** adds an element to the right end of a sequence.

Our model also requires some extensions to the extended body of each routine. Each routine must have an implicitly declared local variable, which is used to count each routine call made in the executable body. If **MAIN** is defined by

```
procedure Main (var B:buf_type)
        unless (cond RoutineError, SpaceError) =
  begin
    var I : integer := 0;

    P(B);
    loop
      Q(B,I+0);         { Second parameter must be const,
                          because I+0 is not a <name expression>.}
      if I = 10 then leave end;
      I := I + 1;
    end;
  end
```

then, the partially expanded definition of **MAIN** looks like

```
procedure Main (var B:buf_type; const MYID:activation_id)
        unless (cond RoutineError, SpaceError) =
  begin
    var Invocation_Count# : integer := 0;
    var I : integer := 0;

    P(B, New_Activation_Id(MYID, Invocation_Count#));
    Invocation_Count# := Invocation_Count# + 1;
    loop
      Q(B, I+0, New_Activation_Id(MYID, Invocation_Count#));
      Invocation_Count# := Invocation_Count# + 1;
      if I = 10 then leave end;
      I := I + 1;
    end;
  end.
```

Thus, the activation_id passed to procedure **P** would be **(SEQ: 0 0)**. The eleven activations of procedure **Q** would have activation_ids **(SEQ: 0 1)** through **(SEQ: 0 11)**. The completely expanded definition of **MAIN** would also have to pass unique activation_ids to each function invocation.

Thus, the simple statement

```
I := I + 2 - 1;
```

with function calls made explicit would expand to

```
I := INTEGER#SUBTRACT( INTEGER#PLUS(I,2), 1);
```

Further, with the activation_ids made explicit, it would look like

```
I := INTEGER#SUBTRACT
      (INTEGER#PLUS(I,2,New_Activation_Id(MYID,Invocation_Count#)),
       1,
       New_Activation_Id(MYID,Invocation_Count# + 1));
Invocation_Count# := Invocation_Count# + 2.
```

Activation_ids are used to describe elements of buffer histories. Since functions cannot have **VAR** parameters, functions cannot affect external buffers. However, functions can use local buffers internally, and so they are passed activation_ids as well as procedures. To avoid an infinite expansion of a routine definition, we will permit some predefined functions that do not take an activation_id parameter. (The predefined function **INTEGER#PLUS** would not normally require an activation_id parameter. In particular, none of the functions required to compute new activation_ids can themselves require activation_id parameters!)[49]

7.3 Operations on ActivatIon_Ids

In later discussions of activation_ids and buffer histories, the following functions will prove useful.

- **Is_Ancestor(A,D)**
  Is the activation identified by activation_id D a descendant of the activation identified by activation_id A? (In other words, is activation_id A an initial subsequence of D?)
- **All_Ancestors(A)**
  Produces a list of the activation_ids of all ancestors of activation identified by activation_id A (i.e., the list of all initial subsequences of A).

7.4 What Is a Buffer?

Gypsy buffers are based on the notion of message buffers as a mechanism to support concurrent process cooperation [Brinch Hansen 73]. Message buffers are first-in/first-out queues. The basic operations insert a message at the end of the queue, and remove a message from the head of the queue. Messages are passed "by value." (Note that Gypsy semantics do not specifically define destructive operations on any data structures. However, the language allows many operations to be optimized to in-place operations to improve efficiency.)

Gypsy has no global variables; all variables in a Gypsy routine must either be declared locally or passed in as parameters. Thus, in order for a routine to have access to a buffer, the buffer must either be declared locally or passed into the routine as a buffer parameter.

Buffers are the only interprocess communication mechanism available to pass information between parallel processes. No other objects are shared between processes that can run in parallel. Gypsy does not include any other process synchronization mechanism (e.g., semaphores, shared variables, monitors,

[49] The use of "+" and "INTEGER#PLUS" in the above example is meant to distinguish uses of "INTEGER#PLUS" that are computing new activation_ids from those in the user's computation.

critical regions).[50] Since all interprocess communication must be done through message buffers, the flow of messages between the two parallel processes completely describes the information flow between them. Gypsy uses just such "buffer histories" to completely describe the interactions between Gypsy routines executing in parallel.

Message buffers can be declared to be of finite, non-negative length. If a length of N is declared, the queue of messages in the buffer cannot be longer than N messages. A buffer is said to be "empty" if the queue has no messages, and is said to be "full" if the queue length is equal to the declared maximum. An attempt to send a message to a full buffer results in the sending process being "blocked" until the send operation can complete. When a message is removed from the buffer (by a receive operation), the sending process can complete the send. Similarly, an attempt to receive a message from an empty buffer results in the receiving process blocking until a message is sent to the buffer. A buffer may have a length limit of zero, in which case the buffer acts like a channel in Hoare's CSP [Hoare 78, Hoare 85], permitting tightly coupled synchronization between processes. The notion of blocking is the basis of Gypsy methods for specifying the behavior of non-terminating, concurrent processes. These methods are discussed later.

7.4.1 Critical Region Model of Buffers

Message buffers can be modeled as queues of messages protected by some mechanism to assure mutual exclusion. For purposes of exposition, and for further investigation of the properties of buffers and buffer histories, we will describe how to model buffer operations using conditional critical regions [Brinch Hansen 73]. We will restrict our discussion to buffers without length restrictions. So far, we can model a buffer as a sequence of messages, a send operation as appending a message on one end, and a receive operation as removing a message from the other end. Figure 7-1 contains a Pascal-like description that illustrates these three aspects of the model. Critical regions are represented by the **USING** statement. Within the **USING** statement, exclusive access to a buffer is assured. The conditional critical region tests a predicate, and only enters the critical region if the predicate is satisfied. If not, it waits until a future time, and tries again. The form is

```
USING B_contents WHEN size(B_Contents) ≠ 0 DO ...
```

This statement waits until **size(B_Contents) ≠ 0**, and then executes the body of the critical region. No other process may alter the value of the buffer between the time the predicate succeeds and the exit of the critical region.

As can be seen, the send operation requires two accesses to the buffer contents, and the receive operation requires three, so exclusive access must be assured lest other processes' buffer operations be interleaved with these accesses. We assume nothing about the scheduling of processes attempting to enter their critical regions, except that scheduling must be fair.

This simple model does not capture the possibility of a finite length restriction on the buffer contents. We will ignore this possibility until we discuss non-terminating processes and blockage specifications.

---

[50]Gypsy parameter passage is defined here essentially as call-by-value-result for non-buffer parameters. Strictly speaking an implementation must make copies of constants at routine calls. However, an implementation might optimize the copying required by call-by-value-result semantics *if* the resulting call-by-reference yields the same semantic interpretation. Passing buffer parameters to parallel processes requires call-by-reference at execution time, so that the single buffer object may be shared. Here we describe the *results* of procedure calls on buffer parameters. We describe the results of procedure calls on buffer histories, and so avoid having to model call-by-reference execution explicitly.

```
B_contents : sequence of msg    { a buffer B of msg }

Using B_contents do
  B_contents := B_contents <: x       { send x to B }

Using B_contents When size(B_contents) > 0 do
  Y := first(B_Contents);        { receive y from B }
  B_contents := nonfirst(B_contents);
end
```

Note: "**<:**" is the Gypsy operator that adds an element onto the right end of a sequence.

Figure 7-1: First Model of Buffer Operations

7.4.2 Primitive Buffer Operations

There are two basic operations on buffers:

- send a message to a buffer, and
- receive a message from a buffer.

These are the only two operations that affect the value of a buffer (i.e., change the queue of pending messages in the buffer). Buffers are created with empty message queues, and there is no way to assign a value to a buffer variable directly.

7.5 Buffer Histories

7.5.1 Functions to Access Buffer Histories

Associated with each Gypsy buffer are histories of all messages that have passed through the buffer. Thus, the history captures the message traffic through a buffer from the start of program execution until some point in the execution. Separate histories record the messages sent to a buffer so far (called the **allto** history), and the messages received from the buffer so far (called the **allfrom** history). The histories are sequences, and the order in which messages are sent to the buffer is preserved. (Recall that the send operation is atomic, so the order in which messages are sent is meaningful even in the presence of concurrent processing.) Since messages are removed from the buffer by a first-in/first-out protocol, the **allfrom** history must be an initial substring of the **allto** history. That is, any messages received must have been received in exactly the order they were sent to the buffer.

Two more aspects of buffer histories that will be needed:

- local versus global histories,
- histories containing "time-stamped" messages.

The distinction between local and global histories is explained below. The need for time-stamps is discussed in the chapter on concurrency.[51]

---

[51]The semantics of buffer parameter passage are straightforward, even in the presence of concurrency.

7.5.2 Global Buffer Histories

Global histories (the **allto** and **allfrom** histories) record all input and output activity using a buffer. The histories are denoted by

allto(B)

and

allfrom(B)

Whenever a **send m to B** operation is performed anywhere in the Gypsy program, the value of the message **m** is added to the end of the **allto** history. Whenever a **receive m1 from B** is performed anywhere in the Gypsy program, the current message at the head of the message queue representing **B** is removed from the queue (and **m1** is assigned its value), and the message is added to the end of the **allfrom** history.

We can now revise our buffer operations model to include updating the global histories as in Figure 7-2.

```
{ Declare a buffer B of msg }
 B : record (contents:sequence of msg;
             allto,
             allfrom:sequence of msg)

 { send x to B }
 Using B do
   B.contents := B.contents <: x;
   B.allto    := B.allto    <: x;
 end

 { receive y from B }
 Using B do
   Y := first(B.Contents);
   B.contents := nonfirst(B.contents);
   B.allfrom  := B.allfrom <: Y;
 end
```

Figure 7-2: Model of Buffer Operations with Global Histories

Note that global histories of non-local buffers (i.e., buffers passed in as parameters) behave as shared variables. To preserve the independence principle in our proofs, we will simulate arbitrary changes to the shared histories of the buffer by inserting statements explicitly updating the global history (and buffer contents) between each pair of statements in the routine. Given the unwinding of function calls performed during normalization, two uses of **ALLTO(B)** in a statement in the original Gypsy program would be broken into two distinct assignments to temporaries. Thus, our newly interspersed statements updating the global buffer histories would cause the expression **ALLTO(B) = ALLTO(B)** to become (in effect) **ALLTO(B) = ALLTO(B#1)**. Thus, the nondeterministic nature of global histories of non-local buffers is accurately reflected in our VCs and proofs.

7.5.3 Local Buffer Histories

Local buffer histories (**INFROM** and **OUTTO**) only reflect message traffic that has been performed by a particular routine activation. Buffer operations performed by the routine's descendants are included in the routine's local history, but operations performed by parent routines or siblings are not. A short example illustrates the basic idea.

Figure 7-3 shows two Gypsy procedures, P and Q. Procedure P calls procedure Q, so let us consider Q first. The body of procedure Q performs two **send** operations, and no **receive** operations. Thus, the local **outto** history of buffer B for procedure Q is the sequence **[3, 5]**, and the local **infrom** history of B with respect to Q is the null sequence. Procedure P also performs two **send** operations, but it also invokes procedure Q. The local **outto** history with respect to P includes not only the two messages that it sends directly, but also the messages that its activation of Q causes to be sent. Thus, the local **outto** history of B in P is the sequence **[2, 3, 5, 7]**. Note that the messages appear in the same order in which they are sent to the buffer. The local **infrom** history of B for P is null.

```
type buf_type = buffer of integer;

procedure P (var B : buf_type) =
  begin
    send 2 to B;
    Q(B);
    send 7 to B;
  end;

procedure Q (var B : buf_type) =
  begin
  var n : integer;

    send 3 to B;
    send 5 to B;
  end;
```

Figure 7-3: Small Example of Procedures and Buffers

In our small example in figure 7-3, the behavior of procedure Q is invariant. Thus, the history of messages sent and received by Q is always the same for each activation of the procedure. In general the behavior of a procedure varies from activation to activation (perhaps depending on the values of some data parameters or messages received from some other processes), and the number and content of messages sent to a buffer can vary from one activation to another. In order to capture this notion in our histories, we must introduce a mechanism for distinguishing the history associated with one routine activation from that associated with another. To do this we use activation_ids. An unique activation_id is associated with each routine activation, and the local histories with respect to a particular activation are selected using the activation_id associated with that activation. Activations_ids are typically denoted by a routine name followed by a "**#**" character and a number. The notation for local histories is

```
infrom(B, Q#1)  == local input history of B
                         with respect to the routine
                         activation identified by "Q#1",

outto(B, Q#2)   == local output history of B
                         with respect to the routine
                         activation identified by "Q#2",
```

where **B** is a buffer, and **Q#1** and **Q#2** are activation_ids.

The manipulation of activation_ids by Gypsy programs is restricted. There are no operations available to Gypsy programmers to construct or decompose activation_ids, or even to compare activation_ids for equality. User defined routines cannot return results of type activation_id, nor is assignment permitted in user programs. The only activation_id available for use in specifying a routine is the implicit formal parameter **myid**. Thus, specification of the procedure can only be written in terms of a single (generic) activation.

VC generation can introduce new activation_ids, representing routines invoked by the procedure. More complete details of the Gypsy procedure call mechanism and activation_ids are discussed in the next section.

7.5.4 Local Buffer Histories as Records of Mappings

All the local buffer histories are initially null sequences. The values of the local histories are altered by the **SEND** and **RECEIVE** statements, and (as we shall see) by extension they can also be altered by procedure calls.

The principle of uniform reference allows us to think of the buffer histories as functions of a buffer, or as record fields in a structured object representing a buffer. The Gypsy Report [Good 85] uses functional notation. The preceding explanation uses record structures. Buffers can be modeled as a record with a field for each type of history (**infrom**, **outto**, **allfrom**, **allto**, and so on) with each of the local history fields being a Gypsy mapping from **Activation_id** to a history sequence. The mapping of **Activation_id** could equally well be placed above a single record structure composed of all the local histories for a particular routine activation.[52]

[52]Another approach to modeling buffer histories is to have a single history sequence representing the global **allto** and **allfrom** histories, and stamp each message with the **activation_id** of the routine doing the **send** or **receive**. Then local histories can be extracted by searching for messages with particular **activation_id** stamps. This requires an operation such as **All_Ancestors**, so that the **send** or **receive** operation can stamp the message with the **activation_id** of all ancestors of the activation, since the message must appear on their local histories. A dual of this is to define a relation **Descendant_of**, to be used when extracting local histories so as to include operations performed by descendants of the given activation.

This alternative approach requires an ontological commitment to globally shared objects. The approach presented in the text was chosen to allow the changes in the histories to be described analogously to other procedure **VAR** parameters.

7.6 Procedure Calls with Buffer Parameters

Basically the procedure call with buffer parameters has the same effect as a primitive **SEND** or **RECEIVE**, except that there may be multiple messages appended to the input and output histories (i.e., the effect of the procedure call is a composition of the individual effects of the components of the invoked procedure body). The procedure call

```
Q(B);
```

would update buffer histories for a local buffer as:

```
Using B do
  B.allto   := B.allto   @ outto(B,Q#2);
  B.allfrom := b.allfrom @ infrom(B,Q#2);
end
```

We have not yet introduced concurrency, so the call of procedure **Q** is effectively a critical region. More generally, if the buffer is not local but is passed to the routine as a parameter, the update rule within the procedure would only refer to the local histories. The buffer updates would be:

```
Using B do
  B.outto(myid)  := B.outto(myid)  @ outto(B,Q#3);
  B.infrom(myid) := B.infrom(myid) @ infrom(B,Q#3);
end
```

The local histories of the parent routine would be updated similarly upon return from this activation. Updating the global histories requires global knowledge about all processes with access to the buffer. In the presence of concurrency, such knowledge is only available for local buffers (i.e., buffers declared as local variables within the routine).

7.7 Writing Specifications about Buffer Histories

Normally specifications are stated in terms of properties of local buffer histories. (You can rarely state anything about the global buffer histories unless the buffer is declared locally, in which case the local buffer histories are identical to the global ones.)

Specifications about a procedure's local histories can appear in external specifications, and are thus exported into proofs of routines calling the procedure. The references to **MYID** in the procedure's external specifications are converted to refer to an appropriate activation_id when the external specifications appear in a VC generated in the proof of a calling routine. We have not yet introduced concurrency, so none of its complications need be addressed yet.

7.8 Summary of Buffers

The functionality and restrictions on Gypsy buffers are summarized below.

A. No assignment operator exists for buffers; only **SEND** and **RECEIVE** change buffer contents and buffer histories.

B. Buffer operations extend histories (i.e., **send** and **receive** operations extend the appropriate input and output history sequences).

C. History extraction functions are well-behaved.

i. Global histories are sequences of every message sent to or received from a buffer.

ii. Local histories are sequences of every message sent to or received from a buffer by a particular activation of a procedure (or any procedure called from it).

iii. Buffer histories have relative-time stamps on messages so that the time-ordered merge of the local histories of parallel processes can be performed deterministically.

D. The functions **`FULL`** and **`EMPTY`** apply to buffers. Since only the state of local buffers is completely determined by a procedure, only the fullness or emptiness of local buffers can be completely determined by a routine's execution behavior; for non-local buffers **`FULL`** and **`EMPTY`** appear to be non-deterministic tests. **`FULL`** and **`EMPTY`** are used mainly to describe instantaneous states in proofs.

E. Global buffer histories and the functions **`FULL`**, **`EMPTY`**, and **`CONTENTS`** represent access to information shared between concurrent processes when applied to non-local buffers. VCs containing references to these functions are constructed to treat their values as if they were nondeterministic.

Chapter 8
CONCURRENCY

Concurrent programming is supported in Gypsy by the **COBEGIN** statement, which invokes several procedures in parallel.[53] The **COBEGIN** is treated almost identically to the normal (sequential) procedure call, with the exception that buffers can be shared between concurrent procedures. This generalization of the sequential procedure call is fairly clean and straightforward. Since buffers are the only objects shared between concurrent procedures, only changes in buffer histories are described during concurrent execution. Gypsy also supports a technique for specifying properties of non-terminating procedures, using the **BLOCK** specification.

8.1 COBEGIN - A Generalized Procedure Call

The **COBEGIN** statement is a strict generalization of the sequential procedure call. The no-aliasing rule is generalized to cover the several procedure calls, and the order of evaluation of the arguments is defined for all the procedure calls in the **COBEGIN**. The **COBEGIN** terminates when all its called procedures terminate (just as in the case of a sequential procedure call).

8.1.1 Generalized Aliasing Restriction

In the sequential case, the no-aliasing rule requires that no actual **VAR** parameter overlap with any other actual parameter in the procedure call. In the concurrent case, this generalizes to cover all the procedure calls under a **COBEGIN**. However, it is relaxed slightly with respect to buffer parameters. The same buffer may be passed as an actual **VAR** parameter to several procedure activations under a single **COBEGIN**. The relaxation only permits the buffer to be overlapped between distinct procedure activations within the **COBEGIN**. Thus, concurrent routines can communicate through the shared buffers, but no dangerous aliasing is permitted within any of the procedure calls under the **COBEGIN**.

In summary, the following constraints apply to the set procedure calls in a **COBEGIN** statement.

- No non-buffer actual **VAR** parameter may overlap with any other actual parameter.
- No buffer actual **VAR** parameter may overlap with another actual parameter in the same routine's parameter list.
- An actual buffer parameter *may* overlap with another actual parameter in a different routine activation -- even if the different activation is of the same procedure (e.g., by means of an

[53]The **AWAIT** statement is a non-deterministic **CASE** statement (poll) of one or more buffer operations. It blocks until one of the buffer operations can complete, and then executes that buffer operation and the corresponding **AWAIT** arm. It is a *sequential* operation, although it is important in supporting effective concurrent programming.

EACH clause).

8.1.2 Effects Observable During Routine Execution

Buffers can be shared between concurrent processes, and so changes in buffers caused by one process can be observed by another concurrent process. The sequential Gypsy model updated **VAR** parameters after the routine call terminated. This was sufficient for buffers in the sequential case, but is not in the concurrent case. Note that this simple model is still sufficient for non-buffer (i.e., non-shared) **VAR** parameters, because no other concurrent process can refer to the altered values of a procedure's non-overlapped **VAR** parameters before the procedure terminates.[54]

8.1.3 Termination of the COBEGIN

Just as in the sequential case, a concurrent procedure call can terminate normally or abnormally.

The **COBEGIN** ***terminates normally*** if all the called routines terminate normally. Execution in the calling routine is suspended until all the called routines have terminated, whereupon execution of the called routine continues. All the non-shared **VAR** parameters are updated when the **COBEGIN** terminates.

The **COBEGIN** ***terminates abnormally*** if all the called routines terminate and one or more of them terminates abnormally. If exactly one of them terminates abnormally, then the appropriate actual condition parameter is signalled in the calling environment. If more than one routine terminates abnormally, then the special condition **MULTIPLECOND** is signalled in the calling environment.

8.2 The BLOCK Specification for Non-Terminating Procedures

For procedure calls in sequential Gypsy, no provision is made for external procedure specifications that are visible during procedure execution. Since none of the procedure's **VAR** parameters are shared, no other active routine can observe the intermediate changes of the procedure; the effects are only observable after the procedure has terminated, and the changes can be observed in the calling environment. In the presence of concurrency this is no longer the case, as mentioned above.

8.2.1 What BLOCK Specifications Mean

Gypsy **BLOCK** specifications provide a technique for specifying the external behavior of processes during their execution. They augment the traditional **EXIT** specification, which specifies effects after the process has terminated. **BLOCK** specifications also allow description of processes that do not normally terminate. The **BLOCK** specification must hold whenever a procedure is blocked waiting to complete a buffer operation. Recursively, a procedure can be blocked waiting for a called procedure to finish a buffer operation.

---

[54]One could argue that use of call-by-reference semantics would lead naturally to immediate reflection of the altered **VAR** parameter values into the calling environment, and that that information could be useful in writing **BLOCK** specifications. However, given that these **VAR** parameters cannot be externally observed (since the calling routine is suspended) until the concurrent processes terminate, this adds no power to either the executable language or the specification language.

8.2.2 Blockage Points

The two primitive buffer operations are **SEND** and **RECEIVE**.

- **SEND** is defined to block or suspend execution of the routine when the destination buffer is full.
- **RECEIVE** is defined to block when the source buffer is empty.

BLOCK specifications are intermittent assertions that must hold whenever a **SEND** or a **RECEIVE** is blocked. Since the **SEND** or **RECEIVE** blocks before completing, the assertion must hold before the buffer operation takes place and the buffer histories are updated. Further, since the operation has blocked, we can assume the "blockage assumption" for the buffer operation (namely, for a **RECEIVE** that the source buffer is full, and for a **SEND** that the destination buffer is empty).

Three other Gypsy constructs are considered to block, as well.

- The **AWAIT** is defined to block if all the guards of the await arms (i.e., the **SEND** and **RECEIVE** forms) are blocked.
- A sequential procedure call is defined to be blocked if the called procedure is blocked, either at a **SEND**, **RECEIVE**, **AWAIT**, or another (sequential or concurrent) procedure call.
- A concurrent procedure call (**COBEGIN**) is defined to be blocked if and only if at least one of its spawned procedure activations is blocked, and all of the spawned procedure activations are either blocked or terminated.

The appropriate blockage assumption for the **AWAIT** statement is a conjunction of the blockage assumption for each of the guards. In the case of a procedure call, no assumption about the states of particular buffers can be made, since the particular construct causing the block is not visible externally. However, the called routine's **BLOCK** specification is visible externally, and must hold if the routine is blocked. Therefore, if a procedure call is blocked, then the appropriately instantiated **BLOCK** specification of each called routine can be assumed.

8.2.3 Effects of Modularity on Provable Buffer Properties

Because of enforced modularity in Gypsy, it is not possible to prove that a buffer operation on a non-local buffer (i.e., a buffer parameter) will not block. To prove that the operation would not block would require knowledge about the global state of the buffer; either that it was not full, or that it was not empty. Since other concurrent, external processes can have access to the buffer, it is not possible to prove this sort of proposition within a Gypsy procedure.[55] Further, since we cannot show that an operation on an external buffer will not block, we are required to verify the blockage specification at *each potential* blockage point. This strong proof method is required so that procedures' proofs depend only on their visible external specifications.

8.2.4 What If a Routine Doesn't Block?

The definition of the blockage specification says nothing about the behavior of the procedure if it does not block. However, knowledge of program behavior can still be derived from the blockage specification.

A. A process can always be "forced to block" by permitting it to execute, but not permitting other processes to send new input or receive output. In the absence of infinite internal chatter, or non-termination of sequential portions of the process, it will ultimately block

[55] It would be possible if the buffer parameter passing mechanism were extended, for example, to pass to a procedure exclusive input or output rights for a buffer.

trying to send or receive to an external buffer.

B. Buffer histories can be extended by a procedure, but previous history information cannot be altered. Blockage specifications often serve as inductive loop invariants, describing ongoing behavior of a procedure. When used in this way as inductive assertions, blockage specifications must be preserved through the body of the loop, because the histories cannot be "fixed up" after the fact.

   Note, however, that use of a blockage specification as a loop invariant permits one "indiscreet" buffer operation. Since the blockage proofs are done before the buffer operation completes, they refer to the buffer history before the relevant **send** or **receive** is done. The blockage specification is only proven of the buffer history after the operation completes, and only if there is another potential blockage point after it. There is no other blockage point if the procedure exits, or enters a non-terminating loop. In either case, the blockage specification is never proven on the final buffer histories. In the case of procedure exit, the exit specification would normally include an assertion about the final buffer histories. The proof methods for proving sequential termination can identify the potential trouble spot for the non-terminating loop.

Contrasted with Process Invariants

Blockage specifications are very similar to process invariants. Both are proven precisely when an externally visible change of state is about to take place. Invariants are usually thought to be proven just after the state change, whereas blockage specifications are thought of as being proven just before the state change. However, since none of the program operations between the state changes are externally visible, these "before" and "after" points appear to be indistinguishable.

Blockage proofs do make an additional assumption compared to simple invariants: they assume the emptiness or fullness of a particular buffer. This permits blockage specifications to be formulated as loop invariants and to distinguish between the various potential blockage points within a loop. A simple example illustrates.

```
procedure pass_it_on (in, out : buffer);
  begin
    block  empty(in) -> outto(out,myid) = infrom(in,myid);
    var m : message;

    loop
      receive m from in;
      send m to out;
    end;
  end
```

The process invariant for **pass_it_on** would have to deal with the two separate blockage points, and the "off by one" lag of the output buffer history.

8.3 Hierarchical Proof of BLOCK Specifications

BLOCK specifications are quite similar to **EXIT** specifications. The **BLOCK** specification of a procedure is introduced into VCs by procedure call statements, just as the **EXIT** specification is. However, whereas the **EXIT** specification is normally associated with an implicit assumption that the procedure terminated, the **BLOCK** specification is associated with an explicit predicate that the procedure blocked.

8.3.1 Blockage Assumption at Procedure Calls

If a sequential procedure call is blocked, then we know that the blockage specification of the called procedure must hold. Blockage of a concurrent procedure call is slightly more complex. A **`COBEGIN`** blocks if at least one of the called procedures is blocked, and the other called procedures are all either blocked or terminated. Thus, the blockage assumption for a **`COBEGIN`** is that for each procedure either its blockage specification or its exit specification holds, and that at least one procedure is blocked, rather than terminated. We introduce the predicate **`IsBlocked`** to identify activation_ids of blocked procedures.

Here is a simple example. For this procedure call

```
cobegin
  P(a,b);
  Q(b,c);
end
```

the blockage assumption would be

```
  if Is_Blocked(P#1)
     then P#Block(a,b)
     else P#Exit(a,b)
  if
& if Is_Blocked(Q#1)
     then Q#Block(b,c)
     else Q#Exit(b,c)
  if
& (Is_Blocked(P#1) or Is_Blocked(Q#1))
```

`P#1` represents the activation_id of this activation of procedure P, and **`P#Block`** and **`P#Exit`** represent the blockage specification and exit specification of **`P`**, respectively. The function argument notation indicates that the specifications are instantiated with the actual parameters of the procedure call. Similar notation is used for procedure **`Q`**.

8.3.2 Buffer States in COBEGIN Blockage

In principle each procedure under the **`cobegin`** can terminate in a different state, and so the different blockage and exit specifications mentioned above might have to be instantiated in different states for the proof to be valid. However, Gypsy's parameter passing rules avoid this difficulty. All the blocked routines share a common state, of course. We need only worry that the terminated routines may have terminated in different states. We observe that:

- Non-buffer **`VAR`** parameters cannot be shared between concurrent procedures.
- Only properties of local buffer histories can be proven within the proof of the individual procedures (and hence only those properties can appear in provable external block or exit specifications).

These observations permit us to ignore the distinctions between the various states on which the procedure specifications need be instantiated, and to use a single state when constructing the blockage assumption. In this single state the value of each non-buffer **`VAR`** parameter is specified by the **`BLOCK`** or **`EXIT`** specification of a single procedure (at most). Further, since provable **`BLOCK`** and **`EXIT`** specifications of the called procedures can only refer to their own local histories, these imported specifications cannot be affected by any state changes caused by the other concurrent routines.

8.4 An Example

Histories and subsequence relations are convenient for specifying some properties for programs. However, some programs are awkward to specify in Gypsy. Consider a distributed version of factorial.

```
function fact (n:integer) : integer =
  begin
    entry n > 0;
    exit result = factorial(n);

    local b:buffer_of_integer;

    send 1 to b;
    cobegin
      each i:[1..n], multiply_by_i (i, b);
    end;
    receive result from b;
    assert   size(allfrom(b)) = n+1
           & size(allto(b)) = n+1
           & last(allfrom(b)) = factorial(n);
  end;

procedure multiply_by_i (i:integer; b:buffer_of_integer);
  begin
    exit   first(outto(b,myid)) = first(infrom(b,myid))*i
         & size(outto(b,myid)  = 1
         & size(infrom(b,myid) = 1;

    var partial:integer;

    receive partial from b;
    send i*partial to b;
  end;

function factorial (n:integer) : integer =
  begin
    entry n > 0;
    exit assume Result = (if n=1 then 1
                                  else n * factorial(n-1) fi);
  end;
```

The correctness of this program depends on some subtle properties of history equality. There is nothing constraining **`multiply_by_i`** from sending a result to **`b`** before reading its "input" from **`b`**. The constraint is merely that the appropriate relation holds between the input and output. The following theorem provides the basis for proving the correctness of such distributed descriptions of essentially sequential computations.

8.4.1 Adequacy of Buffer Histories

The proof in this section shows that Gypsy buffer histories are adequate to prove properties of a broad class of concurrent programs beyond the usual message routers, multiplexers, and filters that are common Gypsy applications. The formulation of this theorem and its proof were assisted by Bob Boyer, J Moore, Natarajan Shankar, and Don Stuart. Their intellectual curiosity and enthusiasm is greatly appreciated.

Consider a Gypsy program with the form of the distributed factorial example. Let the subprocedure compute a commutative function F. The buffer history is a sequence of elements **`[seq: 1, F(1),`**

`F(F(1)), ...]`. The last element of the history is $F^n(1)$. The following theorem shows that this must be true, even though we do not know the order of execution of the concurrent procedure calls under the **`COBEGIN`**.

Theorem:

If *inhistories* is a permutation of $y_1,y_2...,y_n$,
and *outhistories* is a permutation of $F(y_1),...,F(y_n)$,
and $inhistories <: result = 1 :> outhistories$,
and $\forall x,\ \forall\, i > 0,\ F^i(x) \neq x$,

then $result = F^n(1)$.

Recall that the Gypsy operator "**`:>`**" denotes adding one element to the left of a sequence, and that "**`:<`**" denotes adding one element to the right.

Proof: The sequence equality $inhistories <: result = 1 :> outhistories$ yields n+1 equalities of the form:

$$1=z_1$$
$$F(z_{i_1})=z_2$$
$$F(z_{i_2})=z_3$$
$$.$$
$$.$$
$$.$$
$$F(z_{i_n})=result,$$

such that $\forall i$, z_i appears on the right-hand side of only one equality and on the left-hand side of only one equality. Thus all the subscripts i_j are unique, and in the range [1..n].

Now, starting from the equation $F(z_{i_n})=result$, find the one equation which has z_{i_n} on its right-hand side. Call this the candidate equation. Substitute the left-hand side of the candidate equation for its right-hand side in the original equation. Repeat this process, substituting for the z_i term so long as any z_i appears in the expanded form of the original equation.

Observe that an equation can be used as the candidate equation only once. This follows because its right-hand side occurred in the left-hand side of exactly one other equation. It must have been some other equation, because $F(x) \neq x$ by hypothesis. Its right-hand side was introduced by some previous substitution (or it was in the original equation), and it was eliminated by this substitution. Its right-hand side cannot be reintroduced into the expanded equation, because any substitution that would introduce it has already been used and discarded. Thus, the substitution process must terminate.

Further, we must use each of the equations. If we do not use some equation (call it equation A), then there must be a chain of substitutions from the original equation to the equation $1=z_1$ that does not involve equation A. Equation A is of the form $F(z_{i_j})=z_j$, for some j. Thus, the chain of substitutions from the original equation must not ever contain the term z_j, since it could never be removed by a substitution other than equation A. We know that $i_j \neq j$, since $F(x) \neq x$. Further, z_{i_j} could not have been introduced by any substitution in the chain, since it occurs only on the left-hand side of equation A. Thus, there is another equation (call it B) of the form $F(z_{i_{i_j}})=z_{i_j}$, which could not have been used in the chain. Thus, the equation right-hand side $z_{i_{i_j}}$ could not have been used in the chain either. In this manner, we can construct a chain of equations that do not occur in the chain of substitutions.

Since there are only a finite number of equations to start with, our chain of unused equations must be finite. The process of constructing the chain of unused equations can only terminate by using the equation

$1{=}z_1$, or by looping back to another unused equation. It cannot loop back, because that would mean that for some i and j, $F^i(z_j){=}z_j$. Thus, the chain of unused equations must end with $1{=}z_1$. But this is impossible, since the chain of substitutions in the original equation must have ended here (since it terminated, it could not loop.)

Since the substitution process does terminate, we must end up with $result{=}F^i(1)$. **QED**.

The theorem and proof can be generalized for functions $F_1,F_2,...,F_n$ to show that *result* is equal to some sequence of applications of F_i's applied to 1, or any initial value. The restriction that $F^i(x) \neq x$ can also be removed, if a slightly more complex argument is used concerning loops in the substitutions.

8.5 Global Histories as Shared Variables

Buffers can be shared between concurrent processes. Hence, the buffer contents, and the global buffer histories behave as nondeterministic functions, since their values are affected by actions outside the local procedure. We can simulate these nondeterministic changes by explicitly updating the global histories to reflect arbitrary actions by external processes before and after each statement in the procedure body. Since histories grow as extensions of previous histories, these updates must extend the histories appropriately with unknown (possibly empty) message sequences.

8.6 Effects of Concurrent Procedure Calls on Buffers

Since **COBEGIN** is a generalization of the sequential procedure call, the effect of a **COBEGIN** on buffer histories should be a generalization of the effect of a sequential procedure call. This is, indeed, the case. Just as for a normal procedure call, the local buffer histories are updated by appending the messages sent and received by the routines activated by the **COBEGIN**. But this is not enough information. Consider

```
COBEGIN
   P(B);
   Q(B);
END
```

OUTTO(B,MYID) is clearly updated by appending some shuffle of **OUTTO(B,P#1)**, and **OUTTO(B,Q#1)**. The precise ordering of the shuffle depends on the interleaving of the concurrent execution. This could be specified by

```
∃ S,
    is_shuffle(S, [bag: OUTTO(B,P#1), OUTTO(B,Q#1)])
  & OUTTO(B,MYID) = previous_OUTTO @ S
```

where "**[bag: x, y]**" denotes construction of a bag, or multiset. This formulation was used in the proof methods for Gypsy 1.0, and continually dealing with the existential quantifier became awkward in proofs.

The existential quantifier is needed in this formulation because the shuffle of the histories is nondeterministic. We can eliminate the quantifier if we can make the shuffle deterministic. Since the order of the shuffle is determined by the order of interleaving of the **SEND** and **RECEIVE** operations, we choose to extend the buffer histories to include *time stamps*. Each element of the input or output history is paired with an integer value indicating the "time" at which the buffer operation took place. These time stamps need not be chosen to reflect real time, but need only reflect the relative sequence of input and output events. Thus, a buffer history has the invariant property that the time stamps increase monotonically from the earliest element of the history to the latest, and no two distinct elements have the same time stamp. These extended, time stamped local histories are accessed by functions **TIMEDINFROM** and **TIMEDOUTTO**. The extended global histories are accessed by **TIMEDALLFROM** and **TIMEDALLTO**. All four functions take the same arguments as the corresponding non-extended history functions.

Without knowing anything about the actual interleaving of **SEND** and **RECEIVE** operations, we can now specify the incremental change to the buffer history across a **COBEGIN**.

```
TIMEDOUTTO(B,MYID) =   previous_TIMEDOUTTO
                     @ TimedMerge(TIMEDOUTTO(B,P#1),
                                  TIMEDOUTTO(B,Q#1))
```

where **TimedMerge** merges two time-stamped history sequences according to the order of the time stamps. We can do this without loss of generality, because nothing about the order of the time stamps has been specified. The **TimedMerge** expression above has a well-defined value; we simply do not know precisely what it is. Of course, the exit specifications of **P** and **Q** must also hold at this point. Typically, those specifications will state some properties of the local buffer histories of their respective routine activations.

Additional examples and explanation have been published elsewhere [Good 79].

8.7 Treating Blockage as an Exit

So far, the discussion of the effects of procedure calls (both sequential and concurrent) on buffer histories have been described after the procedure calls have completed, in terms of the incremental change to the local history in the calling environment. One appealing aspect of this description is that the **SEND** and **RECEIVE** operations can be specified solely with respect to their effect on the local histories.

Any simple operational model of concurrency requires considerable detail to describe the interleaving of operations under a **COBEGIN**. Although we benefited initially from having a VC generation model based on an operational model, we can now benefit by divorcing the two. Specifically, blockage is treated as a separate kind of "exit" from the **COBEGIN**. The local buffer histories are updated from the histories of the called routines, just as they would be if the **COBEGIN** terminated. This is essentially taking a snapshot of the incremental changes due to the **COBEGIN** up to the point of blockage. These updated histories, along with the **BLOCK** specifications of the called routines, would be used to prove the **BLOCK** specification of the calling routine. The proof of the **BLOCK** specification permits the blockage assumption and the **BLOCK** specifications of the subroutines to be assumed, so a separate path with the appropriate assumptions is generated. Normal symbolic execution of the path yields the required blockage VCs. This pseudo-path does correspond, in some fashion, to the partial execution of the **COBEGIN**.

Chapter 9
PROOF OF ABSTRACT DATA TYPES

The data abstraction mechanism is entirely orthogonal to the execution semantics of Gypsy. Gypsy abstract data types provide a way to enforce independence of proofs and routine definitions from the concrete implementation details of the data types in terms of primitive Gypsy data structures. This independence applies to both executable code and specifications within the program.[56]

Basically, abstract data types provide the three following capabilities in Gypsy:

- hiding concrete data representations through visibility rules for data type definitions, and specifications dealing with the concrete representation of a type,
- proof of concrete data invariants, and
- user-defined equality over abstract types.

9.1 Abstract Type Declarations

A data type declaration denotes an abstract data type, if the declaration includes an "access list" of "privileged units" (procedures, functions, lemmas). The details of the type definition can only be used within the units (functions, procedures, and lemmas) named in the access list. The type definition defines the concrete values of the type. The abstract type name denotes a new, non-decomposable type in contexts in which the concrete details of the type definition are hidden. The privileged units are said to have concrete access to the type, and are called concrete manipulation routines or, sometimes, concrete access functions.

Consider a simple abstract type definition.

```
type stack <push, pop, top, is_empty, empty_stack> =
         record (i:integer[0..max_size];
                 a:array (integer[1..max_size]) of integer);

type array_and_index =
         record (i:integer[0..max_stack_size];
                 a:array (integer[1..max_size]) of integer);
```

The abstract data type **stack** is implemented as an array with an index. Only the privileged routines **push**, **pop**, **top**, and **is_empty** can make use of knowledge of that implementation. Within those

[56]The Gypsy verification environment also applies this to proofs. Details about concrete representations are only permitted in proofs in accordance with the visibility "rights" of the program unit being proven.

routines, the internal definition of type **stack** is visible, and is considered to be a type consistent with any other record structure of similar concrete definition, such as **array_and_index**. In any other context, where the details of type **stack** are not visible, types **stack** and **array_and_index** would not be considered consistent.[57]

9.2 Abstract Versus Concrete Specifications

The external operational specifications of a routine includes the externally visible **ENTRY** and **EXIT** specifications. These are abstract specifications, and cannot decompose objects of abstract types. The header also includes concrete versions of the **ENTRY** and **EXIT** specifications, the **CENTRY** and **CEXIT**. Within the concrete external specifications and within the body of a routine with concrete access, the concrete representation of an abstract data type can be revealed and used.

The **CENTRY** and **CEXIT** specifications are used to factor the proof of the concrete access routine. Using the notation of Hoare axioms, the proof of the routine body can be factored as

```
  ENTRY → CENTRY
& CENTRY { routine_body } CEXIT[58]
& CEXIT → EXIT.
```

Thus, the proof of the **routine_body** can be done entirely at the concrete level, with the proof of the mapping, from abstract to concrete specifications, handled separately.

There is also a **CBLOCK** specification, a concrete analog of the **BLOCK** specification. It is required that

```
CBLOCK → BLOCK.
```

9.2.1 External use of CEXIT and CBLOCK

Assume that two routines, **R1** and **R2**, have concrete access to abstract type **T1**, and that **R1** calls **R2**. Then, whenever the **EXIT** or **BLOCK** specification of **R2** would be used in constructing a VC for **R1**, the **CEXIT** or **CBLOCK** specification is to be used. If **R2** does not have a **CEXIT** specification, then the **EXIT** specification is still used. Similarly, the **BLOCK** is used if **R2** has no **CBLOCK**.

The more general case is that a routine, **R3**, has concrete access to a set of abstract types $\{A_1, A_2, \ldots A_m\}$, and a routine, **R4**, has concrete access to another set of abstract types $\{B_1, B_2, \ldots B_n\}$. The concrete external specifications of **R3** are visible to **R4** when the set of A_i's is a subset of the set of B_j's. In other words, the concrete external specifications of **R3** can be used in the VCs and proof of **R4** when those specifications can only contain the concrete implementation details of abstract data types to which **R4** already has access.

The definition of a specification function can appear in either the **EXIT** or the **CEXIT**. Of course, if both are present they must be consistent, since we prove the formula **CEXIT** $\rightarrow$ **EXIT**. When function definitions are expanded in proofs, the normal rules for visibility of the **CEXIT** apply. The definition from a **CEXIT** specification is used in preference to one in the **EXIT** specification when both are present

[57] Type consistency is required between actual parameters and formal parameters in routine calls. See the Gypsy 2.1 Report, Section 5.9.2, for details on type consistency.

[58] This represents the VCs of the routine body from the **CENTRY** to the **CEXIT**. This notation, reminiscent of Hoare's notation in [Hoare 69], is equivalent to **(generate.vcs (fp extended.routine.body))**, where **extended.routine.body** is the extended routine body using only the concrete external specifications.

and visible.

9.2.2 Lemmas and Data Abstraction

Lemmas about abstract data objects must always be stated without reference to the concrete structure of the type. Thus, the lemma can be used in any context, regardless of visibility of the data representation. If the name of a lemma appears in the access list of an abstract type declaration, it signifies that the *proof* of the lemma can depend on the concrete structure of the type, and that expansion of function definitions in the proof can use the appropriate **CEXIT** forms.

9.2.3 The INITIALLY Specification

All type definitions in Gypsy specify an initial value for objects of that type. Each of the primitive Gypsy data types has a default initial value. If there is no explicit initial value specification in a type declaration, then appropriate default initial values are inherited from primitive constituents of the type. For example, the individual elements of an array of integers would each have the default initial value from type integer (namely, zero). This default initial value can be overridden by an explicit value assignment in the type declaration, such as

```
type my_integer = integer := 13;
type my_array = array (integer[1..max_size]) of my_integer;
```

which declares type **my_array** with elements whose initial values would be 13.

Abstract data type declarations have concrete initial values that are determined in the same way as any other data type declaration. The optional **INITIALLY** specification allows an abstract specification of the initial value. Let us again consider stacks.

```
type stack initially (empty_stack)
     <push, pop, top, is_empty, empty_stack> =
            record (i:integer[0..max_size] := 0;
                    a:array (integer[1..max_size]) of integer);

function empty_stack : stack =
  begin
    cexit   result.i = 0
          & all i:[1..max_size], result.a[i]=0;
    pending;59
  end;
```

This declaration specifies that the initial value of the **i** field of the record is 0, representing an empty stack. The function **empty_stack** is defined, which constructs an empty stack. The declaration of **stack** says that the concrete initial value of an object of type **stack** corresponds to the result of the abstract stack function **empty_stack**. Thus, routines that deal with **stack** at the abstract level can deal with stacks in their initial state. It must be proven that the abstract initial value, identified by the **INITIALLY** specification, and the concrete initial value, given implicitly or explicitly as part of the concrete definition of the type, are equivalent. Thus, in this case we must prove (with full visibility of the concrete structure of stacks) that

```
empty_stack = initial(stack)
```

where **initial(stack)** denotes the concrete initial value for type **stack**. Expanding the equality by

[59] The Gypsy keyword **PENDING** indicates that the executable body of this routine has been left out.

decomposing the record into its two component fields results in the form which must be proven.

```
  empty_stack.i = 0
& all k:integer[1..max_size], empty_stack.a[k] = 0.
```

The function **empty_stack** is parameterless and returns a stack. The Gypsy expression **empty_stack.i** denotes the **i** field of the resulting stack.

The predefined Gypsy function **INITIAL** maps a type name into its default initial value. This is either the abstract initial value or the concrete initial value, depending on whether the concrete structure is visible in the given context. There will not be an explicit abstract representation of the abstract initial value if there is not an **INITIALLY** specification in the type declaration. If an abstract type declaration for type **T** specifies "**INITIALLY (<exp>)**", then we must verify that

```
<exp> = INITIAL(T),
```

where "**INITIAL(T)**" denotes the concrete initial value for **T**. The equality operator denotes abstract equality, if an abstract equality function is defined. Otherwise concrete equality, which is a stronger equivalence relation, is used.

9.3 Proof of Concrete Invariants

Gypsy expresses concrete data invariants with the **HOLD** specification. For example, we can define an abstract type **well_formed_state** that is represented as a **system_state** with the additional constraint that **well_formed** must hold on all values of **well_formed_state**. This constraint is expressed as a restriction on the concrete values of the type **system_state** that are proper values of the type **well_formed_state**. This can be expressed in Gypsy as

```
type well_formed_state <Read_State, Write_State> =
  begin
    S: system_state;
    HOLD well_formed(S);
  end;
```

Note that the **HOLD** specification is defined for the concrete representation of the abstract type.

9.3.1 Proving the HOLD specification

The **HOLD** specification must be proven for the initial value of an abstract type, and whenever a new abstract value is constructed. New abstract values are constructed from concrete values at the boundaries of a concrete access routine. These are the points at which the value ceases to be viewed as a concrete value, and assumes the role of an abstract value. Thus, the **HOLD** specification of a type **T** is added to the **CEXIT** specification of a routine **R** if **R** is included in the access list of **T**.[60] The **HOLD** specification must be instantiated for each of the **VAR** parameters of **R** declared to be of type **T**. For function declarations, the reserved identifier **RESULT**, used to identify the result of a function, must be treated in the same way.

The **HOLD** specification must also be proven when a concrete access routine **R** calls a routine **R1** that has a parameter whose formal type is the abstract type **T**. This is also a "boundary" of **R** at which the concrete value assumes the role of an abstract value. Note that *within* a concrete access routine the concrete

---

[60] This can also be proved by requiring the stronger statement that **CEXIT** $\rightarrow$ **HOLD**. This stronger formulation can have advantages in factoring the proofs, and in forcing the **CEXIT** specification to capture a stronger specification of the routine. The GVE actually produces a VC of this form, with additional hypotheses asserting the **ENTRY** and **HOLD** specifications on the initial values of the parameters.

representation of the abstract value can be manipulated, and the invariant need not hold. Thus, the **HOLD** must be verified whenever a value is passed out of the context of the concrete manipulation routine. This requirement applies to the predefined Gypsy **send**[61] routine, as well as user-defined routines. If a buffer is declared to have elements of an abstract type, then the **HOLD** specification must be proven for any messages sent to the buffer.

9.3.2 Using the HOLD Specification in Proofs

Given that the **HOLD** specification is required to be satisfied at the points described, it can be assumed to be satisfied for any abstract value passed to a concrete manipulation routine. That is, for each formal parameter of **R** declared to be of an abstract type, the **HOLD** for that abstract type instantiated on that parameter name can be assumed along with the **ENTRY** specification in the proof of **R**.[62]

9.4 Equality on Abstract Types

Equality is defined for the primitive Gypsy data types, and equality on structured types (i.e., arrays, records, sequences, sets, and mappings) are defined by appropriately composing the equality relation on the element types. Structured objects are equal if they are type consistent and all the corresponding elements are equal (recursively).[63]

When an abstract data type is defined, no equality relation is automatically available. If equality is needed, an explicit equality function for the abstract data type can be defined. The defining form looks like:

```
function T_equal extends "=" (x,y : T) : boolean = ...
```

The string **extends "="** indicates that this function is intended to define equality for an abstract type. An abstract equality function must be a function of two arguments, both of the abstract type; it must be a boolean-valued function; it must be named on the access list of the abstract data type (so as to have access to the concrete structure of the type). These functions are called equality extension functions or abstract equality functions.

If an abstract type does not have an equality extension function defined for it, then the use of "=" in a context that would call the equality extension (i.e., a context without concrete access) is a call to an undefined function.

Observe that a number of the predefined Gypsy operations implicitly require the use of equality. Such operations as union, intersection, and membership (the Gypsy **in** operator) require that equality be defined for elements of the set or mapping.[64]

---

[61]The procedure call statement for **send** has a special statement form. The **HOLD** specification must be proven when a message is sent to a buffer whose elements contain abstract objects.

[62]This could be handled equally well by adding the instantiated **HOLD** specifications to both the **ENTRY** and **CENTRY**. Thus, the proof **ENTRY** $\rightarrow$ **CENTRY** would still be possible, and the **CENTRY** would not have to restate the **HOLD** specification.

[63]For arrays and records, the elements of the structured value are selected by the regular element selectors, and equality for the element type is used. For sequences and mappings universal quantification is used to range over the appropriate selector expressions (i.e., the range **[1..size(s)]** for a sequence **s**, and the set **domain(m)** for a mapping **m**). For sets defined the quantification is over all members of the set, using the Gypsy membership operator (**IN**).

[64]Details of the consequences of this problem, and the GVE implementation to handle it, are given in ICS Internal Note #209, "Gypsy Data Abstraction", by Larry Akers.

9.4.1 Proof of Equivalence Relation

An equality extension function must be an equivalence relation. That is, it must be symmetric, transitive, and reflexive, and also satisfy the substitution property. The substitution property can be stated as,

> *Substituting an equal value for any term in an expression does not change the meaning of that expression.*

Thus, for every function **F** that has concrete access to the abstract type **T**, it must be proven that

```
all x,y:T,
    all a:T1,
          T_equal(x,y)
        & F_entry(x,a)
        & F_entry(y,a)
       → F(x,a) = F(y,a)
```

F is assumed to take a second parameter of type **T1**. If type **T1** were also an abstract type, it would similarly be necessary to prove

```
all u,v:T1,
    all b:T,
          T1_equal(u,v)
        & F_entry(b,u)
        & F_entry(b,v)
       → F(b,u) = F(b,v),
```

assuming that **T1_equal** is the equality extension for **T1**. These two theorems are sufficient to show that

```
all x,y:T,
    all u,v:T1,
          T_equal(x,y)
        & T1_equal(u,v)
        & F_entry(x,u)
        & F_entry(y,u)
       → F(x,u) = F(y,v).
```

This property also states that all concrete access functions behave as deterministic functions at the abstract level. Thus no concrete access function can reveal information about the concrete value representing an abstract value that would distinguish that value from any other concrete value that is abstractly equal to it. Such information can be revealed, however, by procedures, since procedures are not required to behave deterministically.

9.4.2 Overloading of the Equality Operator (=)

For concrete types the equality operator ("=") is used to denote the appropriate equality function for any particular type. There is no ambiguity in this overloading because the same equality function handles all types that have the same "base type," and the arguments of the equality operator are required to have the same base type.[65]

Within the context of concrete manipulation routines where the concrete structure is visible (the **CENTRY**, **CEXIT**, and body of the routine), the equality operator denotes equality of the concrete data type. The

---

[65] The base type of a type **CT** has the same composition of primitive types as **CT**, but with some size restrictions are removed. See the Gypsy 2.1 [Good 85] manual for details.

abstract equality function can be called explicitly if needed. In any other context, equality operator denotes the abstract equality function. (Of course, use of equality operator to denote abstract equality is not permitted unless an equality extension function has been declared.) Again, there is no ambiguity in this overloading of equality operator.

If a function definition is to be expanded during a proof, care must be taken to assure that the meaning of the equality operator is preserved. Consider the case of expanding the definition of a function **`F`** in the proof of a routine **`R`**. Let **`FADTs`** be the set of abstract data types to which F has concrete access, and **`RADTs`** be similarly defined for **`R`**. The unloading rules fall into three cases:

`FADTs = RADTs`	The equality operator always denotes the same equality function in both **`F`** and **`R`**, so no unloading is necessary. The definition can be taken from the **`CEXIT`** if there is one. If the definition comes from the **`EXIT`**, then the equality operator denotes abstract equality, and any use of equality on an object whose type is in **`RADTs`** must be changed to an explicit call to the abstract equality function.
`FADTs` $\subset$ **`RADTs`**	The definition can be taken from the **`CEXIT`** if there is one. If the definition is taken from the **`EXIT`** specification, then unloading is the same as the previous case. If the definition comes from a **`CEXIT`**, then the equality operator need only be unloaded for those types in **`RADTs`** but not in **`FADTs`**.
Anything else.	The definition must be taken from the **`EXIT`** specification. All uses of the equality operator denote abstract equality, and must be unloaded for all types in **`RADTs`**.

Observe that, in general, for n abstract data types there can be as many as 2^n distinct equality relations. A distinct equality extension function is permitted for each abstract type. The additional equality functions are defined implicitly by composing the semi-abstract (or semi-concrete) equality functions necessary in a context that has concrete access of some subset of the n data types from the abstract equality functions and the definition of concrete equality on the visible portions of the data structure. To see that this is necessary, consider n abstract data types, each with an abstract equality function. Now consider a concrete data type **`CT`**, that is a record structure with n fields, each one being declared as a different one of the n abstract types. Since each of the abstract types has a separate concrete access list, a given routine can be named on any subset of the n abstract types. In each routine the equality operator will denote equality for the concrete record structure, which decomposes into equality for each of the fields of the record. But equality for an individual field can either be abstract equality or concrete equality, depending on whether the routine is named in the concrete access list of the type declaration for the type of that field. This causes no problem, as the equality operator is always unambiguous, and whenever an equality is imported from a specification into a proof, the proper equality relation can be expressed explicitly in the proof context. Several levels of the concrete structure of a type may have to be decomposed explicitly to do this. For example, if **`R1`** and **`R2`** are record structures with two fields, the equality

```
R1 = R2
```

might become

```
  R1.A = R2.A
& Abstract_Equality_Fn(R1.B, R2.B).
```

This observation poses no problem in soundness, but does require some care in doing proofs involving abstract data types.

Chapter 10
PROPOSED LANGUAGE CHANGES

Based on the work done developing the proof methods for Gypsy, some language extensions are proposed. None of these proposals are a radical departure from the existing Gypsy language. The **COUNT** and **MEASURE** specifications extend the power of the specification language. The other proposals merely ease some syntactic restraints, perhaps adding a little clarity to some styles of programming, without adding any new power.

10.1 Additional Assertion Types

User-supplied assertions in Gypsy programs support generation of verification conditions. **ASSERT** statements annotate the procedural body of a routine and, in conjunction with control statements, define paths for verification condition generation. The **ASSERT** statement is the only way a user can specify that some property of the program is to be verified or assumed at a particular point in the program. (The **KEEP**, **HOLD**, and **BLOCK** specifications specify a property to be proven at a number of points in the program.) The key property of the **ASSERT** statement is that it indicates a path break when the linear path segments are constructed.

For this reason, a predicate describing all desired properties of the program state should be present in the current ASSERT statement. (Any information left out of the assertion is not included in verification conditions concerning that program state.) (A simple extension to this type of assertion eases the problem of annotating programs.)

The current **ASSERT** statement comes in the following flavors:

- **ASSERT PROVE <boolean exp> [otherwise C]**
- **ASSERT ASSUME <boolean exp> [otherwise C]**

Both forms of the **ASSERT** statement "break paths" in the verification condition generation process. They differ in whether the boolean expression appears in the conclusion of a verification condition. ("Assumed" assertions only appear as hypotheses in generated verification conditions.)

The proposed extensions are to introduce two new flavors of assertions that do not break paths in the verification condition generation process.

- **PROVE ... [otherwise C]**
- **ASSUME ... [otherwise C]**

These assertions are "proved" or "assumed" during the generation of verification conditions, just as for the **ASSERT** statement. However, since they do not break paths, they provide a new mechanism for the user.

The **ASSUME** statement can be used at a particular point in a routine body to add knowledge concerning the program state at that point. The **PROVE** statement can be used to force a verification condition with a specific conclusion, effectively letting the user split a subgoal from the proof of a verification condition. This mechanism can help the user annotate the program with useful observations, or, allow some program analysis components to insert **PROVE** statements mechanically to construct proofs of various program properties.

The Gypsy optimizer [McHugh 83] is an example of a program analysis component that could make good use of mechanically inserting **PROVE** statements. The optimizer could annotate a program with **PROVE** statements requesting that various overflow and error conditions not occur. In fact the optimizer should generate "optimization conditions" -- "verification condition-like" proposed theorems that should be passed through the proof process, but relate to the optimization of a program, rather than directly to its verification. To draw this distinction, we could extend the **PROVE** statement to accept a string identifying the "type" of proposed theorem to be generated. For example,

```
PROVE "for optimization" ... ;
```

The language definition might include some specific terms, or use the style of the operator extension mechanism of simply accepting a string in an appropriate syntactic position. In this case, the string might tend to be relatively long, and hence somewhat awkward for the user. We might use the default "for verification," and hope that other required **PROVE** assertions are mostly generated mechanically.

10.2 LOOP Count Assertions

In order to facilitate proof of termination of **LOOP** statements, we could extend **ASSERT** statements to include an optional **count** clause, as

```
ASSERT ... [count <non-negative integer expression>];
```

where **<non-negative integer expression>** is constrained to be an **<expression>** that yields a non-negative integer value. Whenever an assertion containing a **count** clause starts a path, it is assumed that the expression yields a non-negative integer value. Whenever an assertion containing a **count** clause ends a path, it must be shown that the expression yields a non-negative integer value. If both the starting and ending assertions of a path contain **count** clauses, then it must be shown that the value of the ending **count** expression is strictly less than the value of the beginning **count** expression.

The **count** clause is limited to **<non-negative integer expressions>** because "less than" is known to be a well-ordering on non-negative integers. If expressions of other types were permitted in the **count** clause, there would have to be an additional mechanism for identifying a well-ordering relation on the type.

10.3 Measure Specification

The measure specification was introduced in section 6.8.4, to facilitate proof of termination of recursive functions. The following describes a simple method sufficient to prove termination of recursive functions in the absence of mutual recursion. It is similar to the mechanism used to prove termination in the Boyer-Moore Theorem Prover [Boyer&Moore 79].

The syntax for **<abstract operational specification>** given in [Good 85] could be extended to include an optional **<measure specification>**, of the form

```
measure <non-negative integer expression>;
```

The value of the **measure** specification instantiated on the routine data parameters must be shown to be a non-negative integer at each call site, including any recursive calls. Further, at recursive call sites, the value of the **measure** specification instantiated on the actual data parameters must be shown to be strictly less than the value of the **measure** specification instantiated on the format parameters of the routine.

10.4 Proper Domain Specification

Recall that satisfying entry specifications is not sufficient to assure proper termination. We now introduce a new specification form precisely to specify the domain (i.e., via a prescriptive entry condition) over which the routine will terminate. Since routines can terminate either normally or abnormally, we need two specification forms:

- **Proper_Domain** - identifies domain over which the function terminates normally.
- **Terminates** - identifies the domain over which the function terminates either normally or abnormally.

These new specification forms support proof that

- terms appearing in specification expressions are well-defined,
- programs terminate,
- programs terminate normally.

This can also be useful to a compiler for suppressing some run-time error checking, since we prove absence of untrapped run-time errors in order to prove normal termination.

10.5 Concrete Lemmas

In the current definition of Gypsy 2.1 the **<non-validated specification expression>** that makes up a lemma body "must be stated in purely abstract terms." [Good 85, section 11.4.4] This proposal would extend the lemma facility to include lemmas about the concrete representation of abstract types.

At present, a lemma body can never refer to the concrete structure of an abstract type. If the lemma name appears in the concrete access list of an abstract type, that indicates that the proof of the lemma can depend on concrete level specifications concerning that type. Statements concerning concrete properties of the abstract type must be stated either in the **HOLD** specification or as **CEXIT'** properties of routines with concrete access to the type definition.

Although this does not represent any restriction in the power of the language (since lemmas are semantically equivalent boolean specification functions), this restriction does have at least two detrimental effects.

a. Lemmas are one of the key mechanisms intended to support modularity in proofs. By disallowing lemmas about concrete structure, we are limiting their use in decomposing proofs about the concrete implementation of abstract data types.[66]

b. The GVE Prover is being extended to convert assumed subgoals to lemmas. Because of the current restrictions on lemmas, subgoals involving concrete specifications of abstract data types cannot be handled.[67]

10.5.1 A Proposed Extension

The proposed extension would add a **`<privileged units>`** list to a lemma declaration, and allow a lemma body to have access to the concrete structure of an abstract type if

- the lemma name appears in the concrete access list of the abstract type,
- the lemma, itself, has a **`<privileged units>`** list, naming those units permitted to use the lemma directly in their proofs, and
- each of the privileged unit named has access to the abstract type.

These points restrict use of the lemma to proofs of routines that already have access to the concrete structure of the type. Thus, we do not introduce any new dependencies by using this lemma in the proof of any of the named routines. (Of course, the top-level proof manager assures that the lemma can only be used in the proofs of units named in the lemmas reference access list.)

The proposed extension mechanism provides a missing level of abstraction within the Gypsy abstract type mechanism. Just as lemmas are a nice syntactic device for phrasing boolean functions with abstract exit specifications, we would like a device for phrasing boolean functions with concrete exit specifications. Concrete lemmas would provide a mechanism for stating properties of the abstract type that might not be satisfied by the unconstrained concrete representation.

10.5.2 An Example

Here is the infamous stack example, making use of a concrete lemma.

---

[66]Again, there is no loss of expressive power. If we restrict abstract data type declarations to take type names, rather than type definitions, then we can always write lemmas about the concrete type. This is yet another example of the need to avoid anonymous types if things are to be uniform.

[67]One future GVE extension being considered would place all VCs in the database as lemmas. This is not generally possible if concrete access to abstract types is not permitted in lemma bodies.

```
type Stack <push, pop, top,
            push_pop_lemma, concrete_push_pop_lemma> =
   begin
        st: record (p : index;
                    a : array [index] of element);
   end;

function push (st:stack, x:element) : stack
                                unless (cond full_err) =
   begin
     exit top(result) = x  and  pop(result) = st';
     pending;
   end;

function pop (st:stack) : stack unless (cond empty_err) =
   begin
     cexit st.p = st'.p-1  and  st.a = st'.a;
     pending;
   end;

    { These 2 lemmas can only refer to stacks as abstract
      objects, though the their proofs may depend on the
      concrete structure of stacks.}

lemma push_pop_lemma (st:stack, x:element) =
   pop(push(st,x)) = st;

lemma top_push_lemma (st:stack, x:element) =
      top(push(st,x)) = x;


      { This lemma can refer to the concrete structure of
        stacks, because its access list restricts it
        sufficiently.}

<push, pop, top, push_pop_lemma>
lemma concrete_push_pop_lemma (st:stack, x:element) =
      pop(push(st,x)).a = st.a with ([st.p+1]:=x)
    & pop(push(st,x)).p = st.p;
```

10.6 User-Specified Semantic Relaxation

The meaning of Gypsy programs sometimes suffers from the problem of being completely specified, even regarding details the programmer considers unimportant. (This failing is common to almost all procedural programming languages, as the programmer must often specify two actions in a particular order, even though no particular order is important.) As described in section 3.6.3, Gypsy program behavior in the presence of abnormal conditions (e.g., arithmetic overflow) is precisely defined. Thus, a Gypsy implementation must take great care either to avoid producing side effects until it is certain that a computation completes successfully, or be prepared to undo any partial side effects of a computation if that computation cannot complete successfully.

An implementation might permit considerable program optimization if it could selectively ignore undoing certain side effects when computations fail to complete. The notion of "user-specified semantic relaxation" allows the user to specify certain variables that can suffer side effects in the presence of abnormal conditions. For example, the statement

```
ignore a,b,c,...;
```

appearing along a path would indicate to the compiler and to the proof methods that no knowledge of the values of the named variables (**a,b,c,...**) was assumed along the rest of the path. All information gathered along the path so far would be left out of any subsequent VCs. The **ignore** statement would be most useful in the body of a condition handler, to indicate that the condition path need not preserve the pristine values of the specified variables.

Similar sort of semantic-relaxation specifications are desirable for purposes of program optimization [McHugh 83], as they specifically grant the optimizer greater freedom to alter the computation into a more efficient one without changing the semantic specification of the program.

Appendix A
Modification History

Modifications to the May, 1986 Edition, incorporated in Version B.

```
  Date                          Item
-------------------------------------------------------------------------------
9-Feb-89        Section 3.6.2-D ("Expression Evaluation"), paragraph 2,
Akers           once began:

The normal result of a Gypsy function must be determined by the
actual parameters of the function call.  The determinacy of the normal exit
case may require proof that concurrent routine calls evaluated within the
evaluation of the function are themselves deterministic.

                This captured neither the notions of nondeterminism arising
                from data abstraction nor the possibility of nondeterministic
                procedures in the implementation prelude.  The paragraph has
                been modified to read:

The normal result of a Gypsy function must be determined by the
actual parameters of the function call.  The determinacy of the normal exit
case may require proof that routine calls evaluated within the
evaluation of the function are themselves deterministic, and that
uses of data abstraction are sound.

-------------------------------------------------------------------------------
9-Feb-89        The second paragraph of Section 3.6.4 ("Dealing with
Akers           Nondeterminism in the Language Definition"), originally
                read:
```

Functions are required to be deterministic. This follows almost directly from easy syntactic checks. Functions are not permitted to:

- call procedures, since procedures need not be deterministic, nor
- invoke concurrent routine calls (which follows from the first premise).

```
                This text was replaced by the following:
```

In order for proofs to be valid, functions are required to be deterministic. Determinacy follows almost directly from easy syntactic checks, though some proof obligation may be imposed. To demonstrate determinacy of a function definition, it is sufficient to show that:

- All called routines are deterministic, and
- The function neither invokes concurrent routine calls (via COBEGIN) nor awaits buffer operations (via AWAIT).

```
-------------------------------------------------------------------------------
9-Feb-89        The last sentence of the first paragraph of Section 4.1,
Akers           "New Statement Forms", has been deleted.  The claim about
                the multiple arity of ASSERT was not true for the
                Gypsy 2.1 language.  The sentence originally read:
```

Since a single ASSERT statement can contain several prove or assume directives, its expansion can include several PROVE statements before the BREAKPATH, and several ASSUME statements after the BREAKPATH.

```
                Furthermore, the following was added to the end of the same
                paragraph:
```

The simpler case is the statement:

```
   ASSERT (ASSUME p);
```

which becomes the canonical form:

```
   ASSUME p;
```

The path is not broken, and no proof is required.

9-Feb-89 Section 4.3.4, "ASSERT", has been extended as follows:
Akers

Having dealt with run-time validation, we can now perform the canonicalization of ASSERT as described in Section 4.1.

9-Feb-89 The paragraph following rule E6 in Section 6.4, "The Axioms
Akers for EP", originally began:

Other equivalent closed-form expressions can yield those component path segments. Let us construct one such closed form. All paths through the body of the loop are broken by ASSERT statements. Hence, all new partial paths resulting from (EP (pair p f) stmt) must begin with an ASSERT statement in stmt. So, the first recursive reference to EP extends these paths that start in stmt through the stmt as the "second iteration" of the loop. All of the normal partial paths from the previous "iteration" become finished this time around (since all paths through the loop are broken by an ASSERT) -- or they become abnormal paths.

It has been modified to read:

Other equivalent closed-form expressions can yield those component path segments. Let us construct one such closed form. All paths through the body of the loop are broken by BREAKPATH statements. Hence, all new partial paths resulting from (EP (pair p f) stmt) must begin with an ASSUME statement in stmt. So, the first recursive reference to EP extends these paths that start in stmt through the stmt as the "second iteration" of the loop. All of the normal partial paths from the previous "iteration" become finished this time around (since all paths through the loop are broken by a BREAKPATH) -- or they become abnormal paths.[68]

9-Feb-89 A new section, 4.3.5, "Alteration Clauses" has been introduced.
Akers

9-Feb-89 The following note was added to the end of Section 4.3.8
Akers (formerly 4.3.7) on "Compact Alteration Clauses".

Applying the syntactic transformation of alteration clauses to function calls to the second example, we have:

```
   gypsy_alter_element# (gypsy_alter_element# (x, i, a), j, b)
```

9-Feb-89 In Section 4.3.17 (formerly 4.3.16) on "LOOP Statement", the
Akers canonicalization of the loop has been changed from "loop" to
"*loop".

[68] The **BREAKPATH** and **ASSUME** must occur because each **LOOP** is required to have a proof-time **ASSERT** statement somewhere along each execution path through its body, and the normalized form of the **ASSERT** yields the **BREAKPATH; ASSUME ...** pair.

```
begin
  *loop
    <loop body>
  end
when
  is *Leave;
end
```

All subsequent references to the canonicalized loop statement have been changed to *loop.

```
----------------------------------------------------------------------------
9-Feb-89        In Section 6.2, "Preliminary Definitions for Manipulating Paths",
Akers           in each of the following definitions, the function "path"
                replaced the function "cons".
```

Definition: **(cons.path P E)** is defined to be **(path (condition P) (cons (StmtList P) E))**, for path P and path element E.

Definition: **(rcons.path P E)** is defined to be **(path (condition P) (rcons (StmtList P) E))** for **P** a path, and **E** a statement.

Definition: **(append.paths P1 P2)** is defined to be

(path (condition P2) (append (StmtList P1) (StmtList P2)))

for paths **P1**, and **P2**.

```
----------------------------------------------------------------------------
9-Feb-89        At the end of Section 6.3, "Tracing Paths: The Functions CPS
Akers           and EP", the following definitions were inserted for
                completeness.
```

Definition: **If p is the pair (pair x y), (Pfirst p) = x and (Psecond p) = y.**

Definition: **If p is the pair (pair x y), (Partial.Paths p) = (Pfirst p) and (Finished.Paths p) = (Psecond p).**

```
----------------------------------------------------------------------------
9-Feb-89        In section 6.4, "The Axioms for EP", a footnote has been added
Akers           to clarify formula E3, as follows:
```

It is not necessary to include f in the second union. It is subsumed by the term ⟨⟨(Finished.paths A)⟩⟩, because

A = (EP (pair p f) <<stmt>>).

Thus set ⟨⟨(Finished.paths A)⟩⟩ includes ⟨⟨f⟩⟩ as a subset, since each equation defining EP preserves the f component of its first argument.

```
----------------------------------------------------------------------------
9-Feb-89        In section 6.4, "The Axioms for EP", the definition of
Akers           Paths.Not.Signalling, which was formerly:
```

Definition: **(Paths.Not.Signalling S name)** ≡
{path | path ∈ S ∧ (condition path) ∉ {name, *NORMAL} }.

has been changed as follows:

Definition: **(Paths.Not.Signalling S name)** $\equiv$
{path | path $\in$ **S** $\wedge$ **(condition path)** $\neq$ **name}.**

Bibliography

[Akers 83] Robert L. Akers.
A Gypsy-to-Ada Program Compiler.
Technical Report 39, Institute for Computing Science, The University of Texas at Austin, December, 1983.

[Anderson 79] R. B. Anderson.
Proving Programs Correct.
John Wiley & Sons, New York, 1979.

[Apt 80] K. R. Apt, N. Francez, W. P. de Roever.
A Proof System for Communicating Sequential Processes.
ACM Transactions on Programming Languages 2(3):359-385, July, 1980.

[Berg 82] H. K. Berg, W. E. Boebert, W. R. Franta, T. G. Moher.
Formal Methods of Program Verification and Specification.
Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Bonyun 82] D. Bonyun, *et al.*.
A Blueprint for a Verification and Evaluation Environment based upon Euclid.
Technical Report FR-5017-82-1, I. P. Sharp Associates, 1982.

[Boyer&Moore 75]
R. S. Boyer, J S. Moore.
Proving Theorems About Lisp Functions.
Journal ACM 22(1):129-144, January, 1975.

[Boyer&Moore 79]
R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer&Moore 81a]
R. S. Boyer and J S. Moore (editors).
The Correctness problem in Computer Science.
Academic Press, London, 1981.

[Boyer&Moore 81b]
R. S. Boyer and J S. Moore.
A Verification Condition Generator for FORTRAN.
In R. S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, 1981.

[Brinch Hansen 73]
Per Brinch Hansen.
Operating System Principles.
Prentice-Hall, 1973.

[Craigen 84] D. Craigen, and M. Saaltink.
Ottawa Euclid and EVES: A Status Report.
In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*. IEEE, 1984.

[Crow 85a] Judith Crow, Dorothy Denning, Peter Ladkin, Michael Melliar-Smith, John Rushby, Richard Schwartz, Robert Shostak, Friedrich von Henke.
SRI Verification System Version 1.8: User's Guide
SRI International Computer Science Laboratory, 1985.

[Crow 85b] Judith Crow, Dorothy Denning, Peter Ladkin, Michael Melliar-Smith, John Rushby, Richard Schwartz, Robert Shostak, Friedrich von Henke.
SRI Verification System Version 1.8: Specification Language Description
SRI International Computer Science Laboratory, 1985.

[Crowe 82] D. Crowe.
Ottawa Euclid Reference Manual.
Technical Report TR-5613-81-7, I. P. Sharp Associates, December, 1982.

[De Millo 79] R. A. De Millo, R. J. Lipton, and A. J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM (5):271-280, 1979.

[Dijkstra 68] E. W. Dijkstra.
A Constructive Approach to the Problem of Program Correctness.
BIT 8-3, 1968.

[DiVito 81] B. L. DiVito.
A Mechanical Verification of the Alternating Bit Protocol.
Technical Report 21, Institute for Computing Science, The University of Texas at Austin, June, 1981.

[DiVito 82] B. L. DiVito.
Verification of Communications Protocols and Abstract Process Models.
Technical Report 25, Institute for Computing Science, The University of Texas at Austin, August, 1982.

[Elspas 72] B. Elspas, *et al.*.
An Assessment of Techniques for Proving Program Correctness.
Computing Surveys 4(2):97-147, June, 1972.

[Erickson 81] Roddy W. Erickson, Editor.
AFFIRM Collected Papers.
Technical Report, USC Information Sciences Institute, February, 1981.

[Floyd 67] R. W. Floyd.
Assigning Meanings to Programs.
In J. T. Schwartz (editor), *Proceedings of Symposia in Applied Mathematics*, pages 19-32. American Mathematical Society, 1967.
Volume 19.

[Gerhart 80] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor and D. S. Wile.
An Overview of AFFIRM: A Specification and Verification System.
In S. H. Lavington (editor), *Information Processing 80*, pages 343-348. October, 1980.
North Holland Publishing Company.

[German 81] Steven M. German.
Verifying the Absense of Common Runtime Errors in Computer Programs.
Technical Report STAN-CS-81-866, Computer Science Department, Stanford University, June, 1981.

[Good 70] D. I. Good.
Toward a Man-Machine System for Proving Program Correctness.
PhD thesis, University of Wisconsin, 1970.

[Good 75] D. I. Good, R. L. London, W. W. Bledsoe.
An Interactive Program Verification System.
In *Proceedings of 1975 International Conference on Reliable Software*. IEEE, 1975.

[Good 78a] D. I. Good, R. M. Cohen, C. G. Hoch, L. W. Hunter, D. F. Hare.
Report on the Language Gypsy: Version 2.0.
Technical Report ICSCA-CMP-10, Institute for Computing Science, The University of Texas at Austin, 1978.

[Good 78b] D. I. Good and R. M. Cohen.
Verifiable Communications Processing in Gypsy.
In *Proceedings of Compcon '78*. IEEE, September, 1978.

[Good 79] Donald I. Good, Richard M. Cohen, James Keeton-Williams.
Principles of Proving Concurrent Programs in Gypsy.
In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 42-52. ACM, New York, 1979.

[Good 82] Donald I. Good.
The Proof of a Distributed System in Gypsy.
In M. J. Elphick (editor), *Formal Specification - Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar*. University of Newcastle upon Tyne, Claremont Tower, Newcastle upon Tyne, NE1 7RU, England., September, 1982.
Also Technical Report #30, Institute for Computing Science, The University of Texas at Austin.

[Good 84] Donald I. Good.
Mechanical Proofs about Computer Programs.
Technical Report 41, Institute for Computing Science, The University of Texas at Austin, March, 1984.

[Good 85] Donald I. Good.
Revised Report on Gypsy 2.1 (Draft).
Technical Report, Institute for Computing Science, University of Texas at Austin, July, 1985.

[Goodenough 75] J. B. Goodenough.
Exception Handling: Issues and a Proposed Notation.
Communications of the ACM 18(12), December, 1975.

[Gordon 79] Michael J. C. Gordon.
The Denotational Descriptions of Programming Languages.
Springer-Verlag, 1979.

[Gries 78] David Gries (editor).
Programming Methodology: A Collection of Articles by Members of IFIP WG2.3.
Springer-Verlag, 1978.

[Hantler 76] S. L. Hantler, and J. C. King.
An Introduction to Proving the Correctness of Programs.
Computing Surveys 8(3):331-353, September, 1976.

[Hare 79] Dwight F. Hare.
A Structure Editor for the Gypsy Verification Environment.
Technical Report 16, Institute for Computing Science, The University of Texas at Austin, August, 1979.

[Hoare 69] C. A. R. Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12(10), 1969.

[Hoare 73] C. A. R. Hoare, N. Wirth.
An Axiomatic Definition of the Programming Language PASCAL.
Acta Informatica 2, 1973.

[Hoare 74] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[Hoare 78] C. A. R. Hoare.
Communicating Sequential Processes.
Communications of the ACM 21(8), August, 1978.

[Hoare 85] C. A. R. Hoare.
Communicating Sequential Processes.
Prentice-Hall International, 1985.

[Hopcroft & Ullman 69]
John E. Hopcroft, and Jefrey D. Ullman.
Formal Languages and Their Relation to Automata.
Addison-Wesley, 1969.

[Howard 76] J. Howard.
Proving Monitors Correct.
Communications of the ACM 19(5), May, 1976.

[King 69] J. C. King.
A Program Verifier.
PhD thesis, Carnegie-Mellon University, 1969.

[Knuth 69] D. E. Knuth, and P. E. Bendix.
Simple Word Problems in Universal Algebras.
In J. Leech (editor), *Computational Problems in Abstract Algebra*. Peramon Press, Elmsford, N.Y., 1969.

[Lampson 77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchel, G. J. Popek.
Report on the Programming Language Euclid.
SIGPLAN Notices 12(2), 1977.

[Levin 81] G. M. Levin, and D. Gries.
A Proof Technique for Communicating Sequential Processes.
Acta Informatica 15:281-302, 1981.

[Levitt 79] Karl N. Levitt, Lawrence Robinson, and Brad A. Silverberg.
The HDM Handbook. Volume 3: A Detailed Example in the Use of HDM.
Technical Report, Computer Science Laboratory, SRI International, June, 1979.

[Locasso 80] R. Locasso, J. Scheid, V. Schorre, P. Eggert.
The Ina Jo Specification Language Reference Manual.
Technical Report TM-(L)-6021/001/00, System Development Corporation, June, 1980.

[London 75] R. London.
A View on Program Verification.
In *International Conference on Reliable Software*. IEEE, April 21-23, 1975.

[London 78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek.
Proof Rules for the Programming Language Euclid.
Acta Informatica (10), 1978.

[Luckham 79] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis.
Stanford Pascal Verifier User Manual.
Technical Report STAN-CS-79-731, Computer Science Departent, Stanford University, March, 1979.

[Luckham 80] D. C. Luckham, and W. Polak.
Ada Exception Handling: An Axiomatic Approach.
ACM Transactions on Programming Languages and Systems 2(2):225-233, April, 1980.

[Majster 79] M. E. Majster.
Treatment of Partial Operations in Algebraic Specifications.
In *Proceedings of the Conference on Specifications of Reliable Software*. IEEE Computer Society, 1979.

[Manna 74] Zohar Manna.
Mathematical Theory of Computation.
McGraw-Hill, 1974.

[McCarthy 63] John McCarthy.
A Basis for a Mathematical Theory of Computation.
In Braffort and Hirschberg (editors), *Computer Programming and Formal Systems*. North Holland, 1963.

[McCarthy 65] John McCarthy, *et al.*.
LISP 1.5 Programmer's Manual.
The M.I.T. Press, Cambridge, Mass., 1965.

[McHugh 83] John McHugh.
Towards the Generation of Efficient Code from Verified Programs.
PhD thesis, The University of Texas at Austin, 1983.

[Milne 76] R. E. Milne and C. Strachey.
A Theory of Programming Language Semantics.
John Wiley, New York, 1976.

[Moriconi 77] Mark S. Moriconi.
A System for Incrementally Designing & Verifying Programs.
PhD thesis, The University of Texas at Austin, 1977.
Also Technical Report #9, Institute for Computing Science, The University of Texas at Austin.

[Musser 77] John V. Guttag, Ellis Horowitz, and David R. Musser.
Some Extensions to Algebraic Specifications.
In *Proceedings of an ACM Conferences on Language Design for Reliable Software*, pages 63-67. ACM, 1977.

[Nixon 85] Mark Nixon.
Private communication.
1985

[Owicki 75] S. Owicki.
Axiomatic Proof Techniques for Parallel Programs.
PhD thesis, Cornell University, Ithaca, N.Y., August, 1975.

[Owicki 76] S. Owicki, and D. Gries.
An Axiomatic Proof Technique for Parallel Programs I.
Acta Informatica 6, 1976.

[Pagan 81] Frank G. Pagan.
Formal Specification of Programming Languages: A Panoramic Primer.
Prentice-Hall, 1981.

[Palme 74] Jacob Palme.
SIMULA as a Tool for Extensible Program Products.
SIGPLAN Notices 9(2):24-40, February, 1974.

[Parnas 72a] D. L. Parnas.
A Technique for Software Module Specification with Examples.
Communications of the ACM 15(5), May, 1972.

[Parnas 72b] D. L. Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM 15(12), December, 1972.

[Polak 80] Wolfgang Heinz Polak.
Theory of Compiler Specification and Verification.
Technical Report STAN-CS-80-802, Computer Science Department, Stanford University, May, 1980.

[Robinson 75] Lawrence Robinson, Karl N. Levit, Peter G. Neumann, Ashok R. Saxena.
On Attaining Reliable Software for a Secure Operating System.
In *Proceedings of the International Conference on Reliable Software*. IEEE, 1975.

[Robinson 77] Lawrence Robinson and Karl N. Levitt.
Proof Techniques for Hierarchically Structured Programs.
Communications of the ACM 20(4):271-283, April, 1977.

[Robinson 79] Lawrence Robinson.
The HDM Handbook. Volume 1: The Foundations of HDM.
Technical Report, Computer Science Laboratory, SRI International, June, 1979.

[Roubine 77] Olivier Roubine, and Lawrence Robinson.
SPECIAL Reference Manual.
Technical Report, SRI International, 1977.

[Schorre 84] Val Schorre, and Judy Stein.
The Interactive Theorem Prover (ITP) User Manual.
Technical Report TM-6889/000/05, System Development Corporation, September, 1984.

[Siebert 84] Ann E. Siebert, and Donald I. Good.
General Message Flow Modulator.
Technical Report #42, Institute for Computing Science, The University of Texas at Austin, March, 1984.

[Silverberg 79] Brad A. Silverberg, Lawrence Robinson,and Karl N. Levitt.
The HDM Handbook. Volume 2: The Languages and Tools of HDM.
Technical Report, Computer Science Laboratory, SRI International, June, 1979.

[Sites 74] Richard Sites.
Proving that Computer Programs Terminate Cleanly.
Technical Report, Computer Science Department, Stanford, May, 1974.

[Smith 80] L. M. Smith.
Compiling from the Gypsy Verification Environment.
Technical Report 20, Institute for Computing Science, The University of Texas at Austin, August, 1980.

[Stoy 77] Joseph E. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
The M.I.T. Press, Cambridge, Mass., 1977.

[Tennent 81] R. D. Tennent.
Principles of Programming Languages.
Prentice-Hall International, 1981.

[Thompson 81] D. H. Thompson, C. A. Sunshine, R. W. Erickson, S. L. Gerhart, and D. Schwabe.
Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models.
Technical Report ISI/RR-81-88, USC/Information Sciences Institute, March, 1981.

[von Henke 75] F. W. von Henke, and D. C. Luckham.
A Methodology for Verifying Programs.
In *Proceedings of 1975 International Conference on Reliable Software*. IEEE, 1975.

[Wirth 73] Niklaus Wirth.
Systematic Programming: An Introduction.
Prentice-Hall, 1973.

Next, for purposes of path tracing, **ASSERT** statements need to be decomposed to the simpler statements, **PROVE**, **BREAKPATH**, and **ASSUME**. The proof directive of the **ASSERT** determines whether or not an execution path is to be broken. The statement

```
ASSERT (PROVE p);
```

is taken as an abbreviation for

```
PROVE p;
BREAKPATH;
ASSUME p;
```

Table of Contents

List of Figures

List of Tables