Spring 5-1986

Generic Specifications in LIL and in Ada via Analogies

George Chester Harrison
Old Dominion University

GENERIC SPECIFICATIONS IN LIL AND IN ADA

VIA ANALOGIES

by

George Chester Harrison
B.A. June 1969, Wilkes College
Ph.D. August 1973, Univeristy of Virginia

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May, 1986

Approved by:

Dar-Biau Liu (Director)

Michael Overstreet

Christian J. Wild

ABSTRACT

GENERIC SPECIFICATIONS IN LIL AND IN ADA
VIA ANALOGIES

George Chester Harrison
Old Dominion University, 1986
Director: Dr. Dar-Biau Liu

We address the problem of making verifiable specifications in generic program units in the Ada Programming Language *. We illustrate the methodologies of LIL proposed by Joseph Goguen and justify the use of such a specification languages using analogy programming originally proposed by Nachum Dershowitz. The work in these areas is new and noticeably incomplete. We address our concern about the reusability of Ada software in a programming environment that includes a specification language like LIL.

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

TO KAY, ALEX, AND NICHOLAS

Acknowledgments

My thanks go to Drs. Hussein Abdel-Wahab, Michael Overstreet and Christian J. Wild for their support and inspiration, to Dr. Janie Jordan, my department chairman at Norfolk State University, who encouraged me to work on this thesis, Dr. Harrison B. Wilson, President of Norfolk State University, for his initial suggestion to pursue this degree and for his providing financial support. Particular and special appreciation goes to my thesis director, Dr. Dar-Biau Liu, who encourages his students to read, understand, and expand on the research of others, and who looks upon these labors with enthusiasm.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Computer science professionals will admit that on occassion they have spent considerable time writing code that either is duplicated elsewhere or at least is very similar to code written for similar problems. In this thesis we address both situations by applying methodologies in recent literature and attempt to relate these methodologies in a unified structure for the Ada Programming Language.

1.1 VERIFIABLE SPECIFICATIONS

We address the problem of reusability, that is, the problem of avoiding reimplementing specifications that have been previously coded. We will demonstrate how to make verifiable specifications in abstract generic program units in Ada utilizing the methodologies proposed in papers by Joseph Goguen (1986), Steven Litvichouk and Allen Matsumoto (1984), and R. M. Burstall and Goguen (1981).

The work in this area is new, and the syntax of the specifications involved have not been standardized and are noticeably incomplete. Thus, we will make little attempt to explain the syntax in detail, and we will take certain minor liberties in our examples.

1.2 ANALOGY PROGRAMMING

For many problems there are analogous solutions. We humans look for analogies to help solve common problems and can understand the usefulness of analogies as an accumulation of experiences. Instead of trying to recall several particular analogies we tend to generalize and abstract by removing unimportant and unessential details and, we hope, keep the valid general principles that controlled the analogies. We use this scheme by creating an instance of the abstraction by giving concrete values and meanings to each essential quality of the analogy. Ada programmers would see this as similar to an instantiation of a generic program unit.

Nachum Dershowitz (1985) addresses this situation, which we will apply here to generic program units in Ada. He proposes an analogy methodology for program development. Finding analogies imply similarities - not equalities. Thus, the generalization produced must be true for all analogous problems, where we define two problems as being analogous when they share the same semantics. This is necessary not only to preserve the essential qualities of the analogies in the abstraction but to allow other analogous problems to be given code from an instantiation of this abstract generic unit; thus, the new problem will have been granted a program code that is both syntatically proper and correct under the assumptions that the original problem

specifications were totally correct and that the input predicate of the abstraction is adequate to guarantee both program correctness and termination.

We will first examine the problem of making verifiable specifications in an Ada software development environment, then illustrate the program development methodology of analogous abstractions, and, finally, illustrate how the verifiable specifications can aid in making analogous abstractions in the Ada environment.

2. REUSABILITY AND SPECIFICATIONS

2.1 REUSABLE SOFTWARE

Ada was initiated and developed by the Department of Defense as an answer to a perceived software crisis brought on, in part, by the complexity and increasing costs of software production and maintenance (DOD 1983). Since the setting of the Ada standard in 1983, there has been considerable discussion and work done on the concept of reusable software especially in the Ada Programming Language. It is hoped that the practice of writing program segments that can be reused will eventually become a quality of good software engineering practices and will address some of the major problems in the crisis of software development.

Reusable software practices seem to be very successful in Japan (Matsumoto 1984); however, the United States seems to be far behind in this art. However, the direction in this country seems to be toward a careful development of programming methodologies that lead to a correct structuring of a complete software development environment using specification languages.

The Honeywell Computer Sciences Center is currently doing research on defining the characteristics of reusable Ada software (St. Dennis et. al. 1986). Among their conclusions are that software packages are reusable if they

were built with reusability as a primary goal, if they are totally correct when used properly, and if the specifications of the packages are clear.

These goals, of course, demand a high degree of abstract programming and an ability to use the tools of making specifications. These specifications should not be so formal as to render the software unreadable nor so informal as to make the programmer question the validity or applications of the packages.

2.2 THE REQUIREMENTS

Although Ada is the only major procedural language that implements abstractions in a generic context, there are needs to make specifications for abstract data types that Ada does not directly support. Many of these requirements are motivated by the desire to produce libraries of Ada components that are to be considered totally correct upon programmer instantiations and are documented in such a structured manner that the instantiated versions can be used with a high degree of confidence.

Our requirements in doing analogy programming also demonstrate the need of a specification language. Dershowitz's analogy methodology for program development is based on finding analogies among program segments, generalizing to form abstract code, and later using the analogies to grant code to analogous problem specifications. In the Ada Programming Language these actions would be to produce a generic unit using properties of analogies, and given a problem specification, to instantiate the correct analogy (generic unit) to produce correct code.

We need to be able to write the specifications for a generic unit so that we can find the analogies without looking into the body of the function, procedure, or package implementation, which may be intentionally hidden from the programmer. Goguen's methodology (1986) aids in this effort.

3. LIMITATIONS OF THE ADA GENERIC CONCEPT

The primary purposes of the generic units in Ada are program factorization and abstraction. Parameterization of the generic units is generally seen as an extension of subprogram parameterization (McGettrick 1982). Actually, the effect of having a generic facility in Ada seems to have had a much greater effect on the theoretical studies of abstract programming and of program specifications than originally anticipated.

Generic units can be written to require the importation of various types, objects, functions, and procedures declared in the Ada environment at the point of instantiation. Although the instantiation may compile

```
generic
     type ELT is private;
     with function "*" (A, B : ELT) return ELT;
     with function "-" (A, B : ELT) return ELT;
function ZERO ( X, Y : ELT ) return ELT;
```

```
function ZERO ( X, Y : ELT ) return ELT is
begin
     return X * Y - Y * X;
end ZERO;
```

Figure 3.1

correctly, additional semantic requirements explicit in the generic body but not the generic specification may not be satisfied, thus, preventing normal termination of the program or producing incorrect outcomes.

For example, if the intention of this odd looking generic function in Figure 3.1 is to return the additive identity in a set "ELT" with well-defined operations "*" and "-", then all works well if ELT is a field with the usual operations of "*" and "-" being instantiated. What is to "prevent" the programmer from instantiating overloaded operators "*" and "-" on a field ELT that do not produce the "desired" result?

A not so obvious problem occurs when the programmer instantiates ELT as a noncommutative ring with the usual "*" and "-" operators. This can occur when ELT is the set of 2 by 2 matrices over some ring R. Certainly, there is no guarantee that the additive identity will be returned by the function "ZERO." Correctness considerations could only be made here by examining the body of the function (the code implementation) which is not always desirable or possible. We will see that this is a symptom of a much larger problem.

We could have annotated the specification part of this code as in Figure 3.2; however, such informality can be disconcerting to the programmer. Even formal, explicit comments that govern the relationships among the types, objects, functions, and procedures in the specification and implementation parts of a generic program unit as in Anna (ANNotated Ada) (Luckham 1985) can be so large that these

```
-----------------------------------------

generic
     type ELT is private
     with function "*" (X, Y : ELT) return ELT;
     with function "-" (X, Y : ELT) return ELT;
     --
     -- ELT is a field with the usual
     --    "multiplication" operation "*" and, if
     --    "+" is the usual "additive" operation,
     --    then a - b = a + (-b), a and b are in
     --    ELT and -b is the usual additive inverse
     --    of b.
     --
function ZERO ( X, Y : ELT ) return ELT;

-----------------------------------------
```

Figure 3.2

comments overtake the number of lines of implemented code; such efforts seem to work well for program verification exercises but do not seem to support the top-down approach necessary in a complete Ada environment that would support the principle of reusability.

Notice also that such comments within this code would have to be rewritten each time a generic unit with the similar requirements is created. Thus, Ada does not support a consistent methodology of supporting the semantics for code requirements. Equivalent specifications written at different times by different people may include varying levels of formality and syntax.

The Ada Programming Language environments provide only syntactic information about the interfaces among compilation units. For promoting reusability it is necessary to provide some support for the semantic interfaces, which current Ada Program Systems Environments (APSE) do not include.

4. THE LIL SPECIFICATION LANGUAGE

Goguen proposes an approach that will integrate the entities needed in an ideal software development system. His methodology provides and extends many semantic ideas: THEORIES, which provide axioms, VIEWS, which describe the interconnections between entities, VERTICAL and HORIZONTAL compositions, which impose semantic structures at different levels and at the same levels of abstraction, respectively, and GENERICS, parameterized software promoting the reusability goal.

He suggests that these items can be best described in his language, LIL (Library Interconnection Language) (Goguen 1986). LIL appears similar to Ada generic specification syntax with formal and informal semantic definitions along with notations that support the notion of semantic binding. Closely tied to these concepts is the language Clear (Burstall and Goguen 1981), which gets much of its power from category theory (Litvichouk and Matsumoto 1984). Again, we will not explain all the syntax involved in LIL and Clear; the reader should consult the excellent sources.

4.1 THEORIES

To formalize data abstractions (sets, variables, functions, abstract data types, etc.) LIL uses theories

containing semantics. Theories may use other theories, etc. to extend their properties and to bind them together. We should visualize LIL theories as the software development system primitives.

For instance, the theory in Figure 4.1.1 describes a single type (set) with no particular semantics. TRIVIAL_SET is to be considered a simple data type void of any algebraic or topological properties.

```
theory TRIVIAL_SET is
     types ELT;
end TRIVIAL_SET;
```

Figure 4.1.1

Although the example in Figure 4.1.2 imports no properties from other theories (except for BOOLEAN from the Ada package STANDARD), it now represents a set and a particular function associated with it. Notice that the function has three axioms binding it with the set.

```
theory PARTIALLY_ORDERED_SET is
     types ELT;
     functions ORD : ELT ELT -> BOOLEAN;
     var E1 E2 E3 : ELT;
     axioms
          ( not(E1 ORD E1) )
          ( if E1 ORD E2 then not(E2 ORD E1) )
          ( if E1 ORD E2 and E2 ORD E3 then
               E1 ORD E3 )
end PARTIALLY_ORDERED_SET;
```

Figure 4.1.2

In LIL the generic concept is embodied in making parameters. In the case of theories this will mean adding structures and semantics to existing theories. So, building on the notion of partially ordered sets we have the WELL_FOUNDED_SET in Figure 4.1.3. The notions of the well-founded set, reduction functions, and end-of-chain functions, mentioned below, are fundamental to proving program termination (Levy 1980), (McGettrick 1982).

```
==========================================

generic theory WELL_FOUNDED_SET
     [ ELT :: PARTIALLY_ORDERED_SET ] is
     vars E1 E2...En... : ELT;
     axioms
          ( if E1, E2,...,En,... is a sequence
            in ELT such that (Ei+1 ORD Ei) for
            i in POSITIVE then the sequence is
            finite )
end WELL_FOUNDED_SET;

==========================================

generic theory REDUCTION_FUNCTION
     [ ELT :: WELL_FOUNDED_SET ] is
     functions REDUCE : ELT -> ELT;
     vars E1 E2 : ELT;
     axioms ( if REDUCE(E1) = E2 then E2 ORD E1 )
end REDUCTION_FUNCTION;

==========================================

generic theory END_OF_CHAIN_FUNCTION
     [ ELT :: WELL_FOUNDED_SET ] is
     functions EOC : ELT -> BOOLEAN;
     vars E1 E2...Ei...En : ELT;
     axioms
          ( for each chain E1 ORD E2 ORD ...
            ORD En in ELT, there is an Ei,
            where i is in 1..n, such that
            EOC(Ei) =  TRUE )
          ( if E1 ORD E2 and EOC(E2) =  TRUE,
            then EOC(E1) = TRUE )
end END_OF_CHAIN_FUNCTION;
==========================================
```

Figure 4.1.3

Next, we introduce in Figure 4.1.4 two general functions and the notion of a generic object. Although these are written without axioms, they will be important in making instantiations.

```
==========================================

generic theory BINARY_OPERATION
     [ ELT :: TRIVIAL_SET ] is
     fuctions BI_OP : ELT ELT -> ELT;
end BINARY_OPERATION;

==========================================

generic theory SINGLE_VARIABLE_FUNCTION
     [ ELT1 ELT2 :: TRIVIAL_SET ] is
     functions FIN : ELT1 -> ELT2;
end SINGLE_VARIABLE_FUNCTION;

==========================================

generic theory OBJECT
     [ ELT :: TRIVIAL_SET ] is
     vars OBJ : ELT;
end OBJECT;

==========================================
```

Figure 4.1.4

4.2 PACKAGES

Packages in LIL contain semantics rather than code; otherwise, note the similarities to Ada style in the LIL package STACK in Figure 4.2.1.

```
generic package STACK [ ELT :: TRIVIAL_SET ] is
     types STACK;
     functions
          PUSH       : STACK ELT -> STACK;
          POP        : STACK     -> STACK;
          TOP        : STACK     -> ELT;
          EMPTY      : STACK     -> BOOLEAN;
          CREATE     :           -> STACK;
     exceptions
          STACK_UNDERFLOW;
          STACK_EMPTY;
     vars
          S : STACK;
          I : ELT;
     axioms
          ( POP(PUSH(S,I)) = S )
          ( TOP(PUSH(S,I)) = I )
          ( EMPTY(CREATE) = TRUE )
          ( EMPTY(PUSH(S,I)) = FALSE )
          ( POP(EMPTY) = STACK_UNDERFLOW )
          ( TOP(EMPTY) = STACK_EMPTY )
end STACK;
```

Figure 4.2.1

The collection of items in the parameter of a generic entity is appropriately called a requirements theory by Goguen. Thus, the requirements theory, unlike generic parameters in Ada, can not only tell us what types, objects, functions, etc. are called for, but can describe the complete semantic primitives necessary to make the package or theory valid. For an interesting instance see Figure

4.2.2, where the functions are pairwise independent in their semantics, and the requirements theory is significantly complex.

```
=======================================

generic package TREE_FUNCTIONS [
     ELT1       :: TRIVIAL_SET;
     ELT2       :: WELL_FOUNDED_SET;
     REDUCE1    :: REDUCTION_FUNCTION[ ELT2 ];
     REDUCE2    :: REDUCTION_FUNCTION[ ELT2 ];
     EOC        :: END_OF_CHAIN_FUNCTION[ ELT2 ];
     FTN        :: SINGLE_VARIABLE_FUNCTION
                        [ ELT2; ELT1 ];
     OP         :: BINARY_OPERATION[ ELT1 ];
     DEFAULT    :: OBJECT[ ELT1 ]
                              ] is

functions
     INO        : ELT2 -> ELT1;
     PREO       : ELT2 -> ELT1;
     POSTO      : ELT2 -> ELT1;

vars
     INPUT      : ELT2;
     RETURN     : ELT1;

axioms

     ( INO(INPUT) = RETURN, where
       if EOC(INPUT) then RETURN := DEFAULT;
       else
          RETURN := INO(REDUCE1(INPUT));
          RETURN :=
               OP( RETURN, FTN(INPUT) );
          RETURN :=
               OP( RETURN, INO(REDUCE2(INPUT) );
       end if; )

     ( PREO(INPUT) = RETURN, where
       if EOC(INPUT) then RETURN := DEFAULT;
       else
          RETURN := FTN(INPUT);
          RETURN :=
               OP( RETURN, PREO(REDUCE1(INPUT)) );
          RETURN :=
               OP( RETURN, PREO(REDUCE2(INPUT)) );
       end if; )
```

```
( POSTO(INPUT) = RETURN. where
  if EOC(INPUT) then RETURN := DEFAULT;
  else
     RETURN := POSTO(REDUCE1(INPUT));
     RETURN :=
          OP( RETURN, POSTO(REDUCE2(INPUT)));
     RETURN :=
          OP( RETURN, FIN(INPUT) );
  end if; )

end TREE_FUNCTIONS;
```

Figure 4.2.2

Notice that the "axioms" in TREE_FUNCTIONS are really the recursive algorithms:

INO -- simulates the inorder movement through a binary tree while doing computations via OP on the nodes of the tree.

PREO and POSTO -- simulate the preorder and postorder movements like INO.

The fact that the parameter type of each function is a well-founded set, that REDUCTION_FUNCTION's are used, and that the BOOLEAN control function, END_OF_CHAIN_FUNCTION, is included guarantee that these recursive functions terminate by the method of structural induction (Levy 1980), (McGettrick 1982).

4.3 VIEWS

The concept of views will allow us to justify how a given LIL entity satisfies a given LIL theory. Suppose for the moment that the package in Figure 4.2.2 contained only the recursive function IND. Note that the requirements theory would be the same and that the axiom for IND would not be any different. Call this package INORDER_COMPUTE. If we wish to realize this generic function as a method of computing the product of floating point numbers in the nodes of a binary tree, we will want a data type called BINARY_TREE_OF_FLOAT defined somewhere in the LIL environment. Since this would be the parameter type of this function, we also need to justify that BINARY_TREE_OF_FLOAT is indeed a well-founded set. Some other items in the requirements theory will need no justification because of the lack of ambiguity. Also, suppose the notion of a proper subtree or just SUBTREE in a BINARY_TREE_OF_FLOAT has been defined (or understood) elsewhere.

```
view B_TREE :: WELL_FOUNDED_SET => BINARY_TREE_OF
     _FLOAT is
     types ( ELT => BINARY_TREE_OF FLOAT )
     ops   ( ORD => SUBTREE )
end B_TREE;
```

Figure 4.3.1

Now B_TREE is an abstract data type with a guarantee that it should be considered a well-founded set with order

relation SUBTREE. Using this view of a binary tree we can now view specific functions dependent on the notion that B_TREE is justifiably a WELL_FOUNDED_SET.

```
==========================================

view TREE_L :: REDUCTION_FUNCTION =>
                    TREE_LEFT is
     types ( ELT => B_TREE )
     ops   ( REDUCE => TREE_LEFT )
end TREE_L;

==========================================

view TREE_R :: REDUCTION_FUNCTION =>
                    TREE_RIGHT is
     types ( ELT => B_TREE )
     ops   ( REDUCE => TREE_RIGHT )
end TREE_R;

==========================================

view NULL_T :: END_OF_CHAIN_FUNCTION =>
                    NULL_TREE is
     types ( ELT => B_TREE )
     ops   ( EOC => NULL_TREE )
end NULL_T;

==========================================

generic view TREE_N :: SINGLE_VARIABLE_FUNCTION =>
     TREE_NODE[ ELT :: BINARY_TREE_OF_FLOAT;
                X   :: TRIVIAL_SET ] is
     types ( ELT => B_TREE )
     ops   ( FTN => TREE_NODE )
end TREE_N;

==========================================
```

Figure 4.3.2

4.4 INSTANTIATIONS

We now have enough semantic information and semantic justifications to make an instantiation of the function INORDER_COMPUTE. LIL utilizes the make command using views of theories and packages, and if natural defaults exists, make uses the theories themselves.

```
=========================================

make BINARY_TREE_PRODUCT is INORDER_COMPUTE
     [ FLOAT; B_TREE; TREE_L; TREE_R;
       NULL_T; TREE_N; "*"; 1.0 ]        end

=========================================
```

Figure 4.4.1

To further illustrate the concepts thus far we will build examples on systems of algebraic structures:

```
=========================================

theory MONOID is
     types M;
     functions * : M M -> M (assoc, id : 1);
end MONOID;
--
-- assoc implies that the function * satisfies
--    (M1 * M2) * M3 = M1 * (M2 * M3)
-- id : 1 implies that the function * satisfies
--    M1 * 1 = M1 = 1 * M1

=========================================

generic theory GROUP[ M :: MONOID ] is
     functions * : M M -> M (assoc, inv, id : 1);
end GROUP;
--
-- inv implies that for each M1 there is M1inv
--    such that M1 * M1inv = 1 = M1inv * M1
-- the notations of assoc and id : 1 are imported
--    from MONOID and are included for completeness

=========================================
```

```
generic theory ABELIAN_GROUP[ G :: GROUP ] is
     functions * : M M -> M
                   (assoc, comm, inv, id : 1);
end ABELIAN_GROUP;
--
-- comm implies that M1 * M2 = M2 * M1

=========================================

generic theory RING[ A :: ABELIAN_GROUP ] is

     generic view A+ :: RING =>
          ABELIAN_GROUP[ A :: GROUP ] is
          ops ( * => + )
     end A+;
     --
     -- because of the "traditional" view of
     --    using + instead of * as the operator
     --    in the underlying abelian group in a
     --    ring we use this generic view to change
     --    its notation before using * as the
     --    "multiplicative" RING operator
     --
     functions
          + : A A -> A (assoc, comm, inv, id : 0);
          m : A -> A;
          - : A A -> A;
          * : A A -> A (assoc);
     vars A1 A2 A3 : A;
     axioms
          ( (A1 + A2)*A3 = (A1*A3) + (A2*A3) )
          ( A1*(A2 + A3) = (A1*A2) + (A1*A3) )
          ( mA1 = A1inv )
          ( A1 - A2 = A1 + (mA2) )
end RING;

=========================================

generic theory FIELD[ R :: RING ] is
     functions
          + : R R -> R (assoc, comm, inv, id : 0);
          m : R -> R;
          - : R R -> R;
          * : R R -> R (assoc, comm, id : 1);
          / : R R -> R;
     vars R1 ONE_OVER_R1 R2 : R;
     axioms
          ( if R1 /= 0, then there is
            ONE_OVER_R1 such that R1 * ONE_OVER_R1
            = 1 = ONE_OVER_R1 * R1 )
          ( R2 / R1 = R2 * ONE_OVER_R1 )
end FIELD;
=========================================
```

Figure 4.4.2

The following package contains the necessary axioms to produce a two element field isomorphic to the field Z2 = {0,1}; we will then be able to instantiate the data type (set) BOOLEAN with new operators to make a new field based on {FALSE, TRUE} with different operators, of course.

```
==========================================

package ZEE2 is
     types Z;
     vars Z0 Z1 : Z;
     functions
          PLUS  : Z Z -> Z
                  (assoc, comm, inv, id : Z0);
          TIMES : Z Z -> Z
                  (assoc, comm, id : Z1);
     axioms
          ( Z1 PLUS Z1 = Z0 )
          ( Z0 TIMES Z0 = Z0 )
          ( Z1 TIMES Z0 = Z0 )
end ZEE2;

==========================================

view BOOLEAN_SET :: ZEE2 => BOOLEAN is
     types ( Z => BOOLEAN )
     ops   ( PLUS  => NOT(EXOR) )
           ( TIMES => AND )
end BOOLEAN_SET;

==========================================

make BOOLEAN_FIELD is FIELD[ BOOLEAN_SET ] end

-- The usual set of BOOLEAN = {FALSE, TRUE} has
--    now been given a field structure.

==========================================
```

Figure 4.4.3

4.5 COMPOSITION

Goguen introduces two fundamental programming activities - horizontal and vertical. Horizontal activities alter structures at a fixed level of abstraction; whereas, vertical activities transform entities into other entities among levels of abstraction while preserving semantics. One particular vertical activity is illustrated in Figure 4.5.1 as an example of composition.

```
generic package STACK_FUNCTIONS[ F :: FIELD ]
     needs STACKF :: STACK[F] is

     functions

          ADD    : STACK -> STACK;
               -- add top two elements in stack
               --   and push sum

          SUM    : STACK -> STACK;
               -- subtract top two elements in
               --   stack and push difference

          MUL    : STACK -> STACK;
               -- multiply top two elements in
               --   stack and push product

          DIV    : STACK -> STACK;
               -- divide top element by next
               --   element in stack and push
               --   quotient

          MINUS  : STACK -> STACK;
               -- change sign of top of stack

     exceptions
          ZERO_DIVIDE;

     vars
          S1 S2 S3 : STACK;
          F1 F2    : F;
```

```
axioms

     ( F1 = TOP(S1) )  --  notation
     ( S2 = POP(S1) )
     ( F2 = TOP(S2) )
     ( S3 = POP(S2) )

     ( ADD(S1) = PUSH(S3, F1 + F2) )
     ( SUB(S1) = PUSH(S3, F1 - F2) )
     ( MUL(S1) = PUSH(S3, F1 * F2) )
     ( DIV(S1) = PUSH(S3, F1 / F2) )
     ( MINUS(S1) = PUSH(S2, mF1) )
     ( if F2 = 0 then DIV(S1) = ZERO_DIVIDE )

end STACK_FUNCTIONS;
```

Figure 4.5.1

In this generic package we must provide STACK_FUNCTIONS with STACKF, a version of package STACK with elements having field characteristics. Since the requirements theory in STACK demands TRIVIAL_SET (see Figure 4.2.1), using a field here which is more specific than TRIVIAL_SET is proper. STACK can have several generic bodies in LIL. To choose one of these versions, say STACK_BODY_3, instantiate as in Figure 4.5.2.

```
make STACK_FUNCTIONS[F] needs STACKF =>
                              STACK_BODY_3[F]  end
```

Figure 4.5.2

Or, we can do the same for a specific field demonstrating the use of horizontal instantiation (FIELD to FLOAT) and vertical instantiation as in Figure 4.5.3.

```
make FLOAT_STACK_FUNCTIONS is
     STACK_FUNCTIONS[ FLOAT ] needs
          STACKF => STACK_BODY_3[ FLOAT ]  end
```

Figure 4.5.3

5. THE METHODOLOGY OF MAKING ANALOGIES

5.1 INTRODUCTION

Our first step in the process of making generalizations is to find an analogy between the final assertions (problem specifications) in the program segments in two similar problems. Next we may apply an abstract mapping of one of the program segments to the generalized program segment resulting in the generic abstraction.

Great care must be taken in making these abstractions: certain preconditions can be lost, rendering the abstraction impossible to prove correct. However, the essential parts of the lost preconditions may in fact be recovered when we use the method of weakest preconditions (Dijkstra, 1976).

Finally, once the correctness has been demonstrated other problems with analogous specifications to those of the abstraction may be granted correct code by assigning the elements of the concrete problem specification to the elements of the generalized scheme by way of an instantiation mapping.

5.2 TWO SIMILAR FUNCTIONS

We examine some of the implications of this methodology to only one particular area of the Ada Programming Language - generic functions - using recursive functions in Figures 5.2.1 and 5.2.2.

```
type CELL;
type BINARY_TREE is access CELL;
type CELL is
     record
          LEFT  : BINARY_TREE;
          NODE  : FLOAT;
          RIGHT : BINARY_TREE;
     end record;

function INORDER_PRODUCT ( TREE : BINARY_TREE )
                         return FLOAT is
--
-- This function finds the product of all numbers
--    in each node (of type FLOAT) in the binary
--    tree.
--
     PRODUCT : FLOAT := 1.0;

begin -- INORDER_PRODUCT

     if TREE /= null then
          PRODUCT := INORDER_PRODUCT( TREE.LEFT );
          PRODUCT := PRODUCT * TREE.NODE;
          PRODUCT := PRODUCT *
               INORDER_PRODUCT( TREE.RIGHT );
     end if;
     return PRODUCT;

end INORDER_PRODUCT;
```

Figure 5.2.1

```
-----------------------------------------

function FIBONACCI ( P : POSITIVE )
                return INTEGER is
--
-- This function finds the P-th term of the
--    Fibonacci sequence:  F(P) = F(P-1) + F(P-2)
--    for P greater than 2, where F(1) = F(2) = 1
--
     F : INTEGER := 1;

begin -- FIBONACCI

     if P > 2 then
          F := FIBONACCI( P-1 ) +
               FIBONACCI( P-2 );
     end if;
     return F;

end FIBONACCI;

-----------------------------------------
```

Figure 5.2.2

5.3 ANALOGIES

In searching for possible analogies between these two functions observe the following: Both functions are controlled by a single boolean expression, do not possess loops, have a single local return variable with a default value utilized at the lowest recursive call, and have a single parameter. The function calls are not compositions. The type of the parameter and the type of the result essentially have no direct operational relationship.

To preserve the integrity of the algorithms, to allow for a greater level of abstraction of the parameter type, and to allow for binary operations which may be non-commutitive or non-associative, we will present the abstraction using three primary statements (as in INORDER_-PRODUCT). To begin to make formal analogies we present the following adjusted functions:

```
-----------------------------------------

function INORDER_PRODUCT ( TREE : BINARY_TREE )
                  return FLOAT is
     PRODUCT : FLOAT;
begin -- INORDER_PRODUCT
     if (TREE = null) then
          PRODUCT := 1.0;
     else
          PRODUCT := INORDER_PRODUCT( TREE.LEFT );
          PRODUCT := PRODUCT * TREE.NODE;
          PRODUCT := PRODUCT *
               INORDER_PRODUCT( TREE.RIGHT );
     end if;
     return PRODUCT;
end INORDER_PRODUCT;

-- AND
```

```
Fucntion FIBONACCI ( P : POSITIVE )
                  return INTEGER is
     F : INTEGER;
begin -- FIBONACCI
     if ( N in 1..2 ) then
          F := 1;
     else
          F := FIBONACCI( P-1 );
          F := F + 0;
          F := F + FIBONACCI( P-2 );
     end if;
     return F;
end FIBONACCI;
```

Figure 5.3.1

```
#################################################################

ANALOGIES                                   ABSTRACT TO

types
  BINARY_TREE >>>>> POSITIVE      ==> T
  FLOAT >>>>>>>>>>> INTEGER       ==> Z

relations
  (TREE = null) >>> (N in 1..2) ==> K : T -> BOOLEAN
  * >>>>>>>>>>>>>>> +            ==> BINARY_OP :
                                       Z x Z -> Z

objects of type Z
  1.0 >>>>>>>>>>>> 1             ==> DEFAULT
  PRODUCT >>>>>>>> F             ==> RETURN_VALUE

objects of type T
  TREE >>>>>>>>>>> P             ==> INPUT_VALUE

function of type Z value
  TREE.NODE >>>>>>> 0            ==> MIDDLE_FTN(INPUT_VALUE)

functions of type T value
  TREE.LEFT >>>>>>> P-1          ==> FIRST_FTN(INPUT_VALUE)
  TREE.RIGHT >>>>>> P-2          ==> LAST_FTN(INPUT_VALUE)

#################################################################
```

Figure 5.3.2

5.4 A GENERIC RECURSIVE FUNCTION

We need some guarantees that our abstract function terminates. Looking towards structural induction we will make some assumptions on the Boolean function K and on the functions of type T value: FIRST_FTN and LAST_FTN (Figure 5.4).

Notice the great similarities between this Ada generic function RECURSIVE_ANALOGY (Figure 5.4) and the LIL recursive function INO (Figure 4.2.2):

```
   Ada                              LIL

type T..                  <=>   ELT2 :: WELL_FOUNDED_SET..
function FIRST_FTN..      <=>   REDUCE1 :: REDUCTION_FUNCTION..
function LAST_FTN..       <=>   REDUCE2 :: REDUCTION_FUNCTION..
function K..              <=>   EOC :: END_OF_CHAIN_FUNCTION..
type Z..                  <=>   ELT1 :: TRIVIAL_SET..
function MIDDLE_FTN..     <=>   FTN :: SINGLE_VARIABLE_FUNCTION..
DEFAULT..                 <=>   DEFAULT :: OBJECT..
function BINARY_OP..      <=>   OP :: BINARY_OPERATION..
     .....                              .....
RECURSIVE_ANALOGY         <=>   INO
     .....                              .....
INPUT_VALUE               <=>   INPUT
RETURN_VALUE              <=>   RETURN
     .....                              .....
Implementation            <=> Axiom
```

```
-----------------------------------------

generic
     type T is private;
     with function FIRST_FTN (INPUT_VALUE : T)
                              return T;
     with function LAST_FTN  (INPUT_VALUE : T)
                              return T;
     with function K         (INPUT_VALUE : T)
                              return BOOLEAN;
     --
     -- T must be a well-founded set such that
     --   i) objects FIRST_FTN(INPUT_VALUE) and
     --   LAST_FTN(INPUT_VALUE) are less than
     --   INPUT_VALUE,
     --   ii) there is an element in T, nil,
     --   such that K(nil) = TRUE, and
     --   iii) if x and y are in T such that x
     --   is less than y and K(y) = TRUE, then
     --   K(x) = TRUE.
     --
     type Z is private;
     with function MIDDLE_FTN (INPUT_VALUE : T)
                               return Z;
     DEFAULT : in Z;
     with function BINARY_OP  (U, V : Z) return Z;

function RECURSIVE_ANALOGY ( INPUT_VALUE : T)
                    return Z;
--------
function RECURSIVE_ANALOGY ( INPUT_VALUE : T)
                    return Z is
     RETURN_VALUE : Z;
begin -- RECURSIVE_ANALOGY

     if K( INPUT_VALUE ) then
          RETURN_VALUE := DEFAULT;
     else
          RETURN_VALUE :=
               RECURSIVE_ANALOGY(
                    FIRST_FTN(INPUT_VALUE) );
          RETURN_VALUE :=
               BINARY_OP( RETURN_VALUE,
                    MIDDLE_FTN(INPUT_VALUE) );
          RETURN_VALUE :=
               BINARY_OP( RETURN_VALUE,
                    RECURSIVE_ANALOGY(
                         LAST_FTN(INPUT_VALUE) ) );
     end if;
     return RETURN_VALUE;

end RECURSIVE_ANALOGY;
-----------------------------------------
```

Figure 5.4

There is a very natural transformation between the LIL package and the Ada generic function. The correctness in both instances, of course, depends on programmer instantiation; however, these considerations depend on the comments in the Ada specification part of the generic function, which are written in informal mathematical logic, at best. On the other hand, the governing specifications in the LIL package are based in the requirements theory whose carefully written primitives could be viewed with simple commands in an Ada environment. Also these primitives govern "all" the software written in that environment, thus, giving consistency and the quality of reusability to the entire system.

Moreover, in Ada we will essentially have to examine the implementation of the generic function to be able to make analogies; however, in LIL the axioms in the specifications correspond to the main algorithms in the generic package. Therefore, the programmer need only examine the LIL specifications for analogies. We shall also see the importance of having a single specification in LIL that can be linked to several alternate implementations by means of different views; whereas in Ada each generic specification unit corresponds to exactly one implementation unit.

5.5 A GENERIC ITERATIVE FUNCTION

Using the methods of Colussi (1984) there is a natural program transformation from the recursive inplementation of INORDER_COMPUTE to an iterative version. However, since FIBONACCI is a second-order difference equation, a natural program transformation maps to a function in closed form; so, the program transformations drop the analogies.

Therefore, we must preserve the meaning and intent of all the abstract types, functions, and objects to make a true transformation from the recursive abstraction to an iterative abstraction. We might do well in simulating the operation of the production of stacks of activation records.

```
-----------------------------------------
generic
     (generic SPECIFICATION exactly the same as
          RECURSIVE_ANALOGY)
function ITERATIVE_ANALOGY ( INPUT_VALUE : T )
          return Z;
-----------------------------------------

-- For the body of ITERATIVE_ANALOGY we will use
--    the following generic package:

generic
     type ITEM is private;
package STACK is
     procedure PUSH ( X : in ITEM );
     procedure POP  ( X : out ITEM );
     function STACK_EMPTY return BOOLEAN;
end STACK;

-----------------------------------------
```

```
-----------------------------------------

package body STACK is
     type CELL; type ELEMENTS is access CELL;
     type CELL is record
          NODE : ITEM;
          NEXT : ELEMENTS;
       end record;
     S : ELEMENTS := null;

     procedure PUSH ( X : in ITEM ) is
          TOP : ELEMENTS := new CELL'(X,S);
     begin -- PUSH
          S := TOP;
     end PUSH;

     procedure POP ( X : out ITEM ) is
     begin -- POP
          X := S.NODE;
          S := S.NEXT;
     end POP;

     function STACK_EMPTY is
     begin -- STACK_EMPTY
          return S = null;
     end STACK_EMPTY;
end STACK;

-----------------------------------------

function ITERATIVE_ANALOGY ( INPUT_VALUE : T )
          return Z is

     RETURN_VALUE : Z;

     type CALL_TYPE is (FIRST,LAST);

     type ACTIVE is record
          CALL : CALL_TYPE;
          TEE  : T;
          RET  : Z;
       end record;

     REC : ACTIVE;

     package ACTIVE_STACK is new STACK (
                    ITEM => ACTIVE );
          use ACTIVE_STACK;
```

```
        procedure PUSH_LIST ( N : in T ) is
             R : ACTIVE;
        begin -- PUSH_LIST
             if not K(N) then
                  R := (FIRST, N, DEFAULT);
                  PUSH( R );
                  PUSH_LIST( FIRST_FTN( N ) );
             end if;
        end PUSH_LIST;

        procedure CLEAN_UP ( R : Z ) is
        begin -- CLEAN_UP
             RETURN_VALUE := BINARY_OP(
                  R, MIDDLE_FTN(REC.TEE) );
             if K(LAST_FTN(REC.TEE)) then
                  RETURN_VALUE :=
                       BINARY_OP(RETURN_VALUE,
                            MIDDLE_FTN(REC.TEE));
             else
                  REC := (LAST,REC.TEE,RETURN_VALUE);
                  PUSH( REC );
                  PUSH_LIST( LAST_FTN(REC.TEE) );
             end if;
        end CLEAN_UP;

   begin -- ITERATIVE_ANALOGY

        if K( INPUT_VALUE ) then
             return DEFAULT;
        else
             PUSH_LIST( INPUT_VALUE );
             while not STACK_EMPTY loop
                  POP( REC );
                  if K( FIRST_FTN( REC.TEE ) and
                            REC.CALL = FIRST then
                       CLEAN_UP( REC.RET );
                  elsif REC.CALL = FIRST then
                       CLEAN_UP( RETURN_VALUE );
                  else
                       RETURN_VALUE := BINARY_OP(
                            REC.RET, RETURN_VALUE );
                  end if;
             end loop;
             return RETURN_VALUE;
        end if;

   end ITERATIVE_ANALOGY;
```

Figure 5.5

6. INSTANTIATING ANALOGIES

6.1 ITERATIVE INSTANTIATIONS

Suppose now that we wished INORDER_PRODUCT and FIBONACCI to be iterative instead of recursive. We need only grant the names, types, relations, and objects meaning in the iterative abstraction. Thus, we make the following instantiations in Figures 6.1.1 and 6.1.2:

```
#################################################################

function NULL_TREE ( TREE : BINARY_TREE ) return BOOLEAN is
begin
     return TREE = null;
end NULL_TREE;

function TREE_NODE ( TREE : BINARY_TREE ) return FLOAT is
begin
     return TREE.NODE;
end TREE_NODE;

function TREE_LEFT ( TREE : BINARY_TREE )
                                 return BINARY_TREE is
begin
     return TREE.LEFT;
end TREE_LEFT;

function TREE_RIGHT ( TREE : BINARY_TREE )
                                 return BINARY_TREE is
begin
     return TREE.RIGHT;
end TREE_RIGHT;
```

```
function INORDER_PRODUCT is new ITERATIVE_ANALOGY (
          T                => BINARY_TREE,
          FIRST_FTN        => TREE_LEFT,
          LAST_FTN         => TREE_RIGHT,
          K                => NULL_TREE,
          Z                => FLOAT,
          MIDDLE_FTN       => TREE_NODE,
          DEFAULT          => 1.0,
          BINARY_OP        => "*"                          );
##################################################################
```

Figure 6.1.1

Notice the first four functions in Figure 6.1.1 give the names needed in the LIL views in Figures 4.3.1 and 4.3.2, and the instantiation, above, corresponds to the LIL instantiation in Figure 4.4.1.

```
##################################################################
function INITIAL_VALUE ( P : POSITIVE ) return BOOLEAN is
begin
     return P in 1..2;
end INITIAL_VALUE;

function ZERO ( P : POSITIVE ) return INTEGER is
begin
     return 0;
end ZERO;

function MINUS_ONE ( P : POSITIVE ) return POSITIVE is
begin
     return P-1;
end MINUS_ONE;

function MINUS_TWO ( P : POSITIVE ) return POSITIVE is
begin
     return P-2;
end MINUS_TWO;

function FIBONACCI is new ITERATIVE_ANALOGY (
     T                => POSITIVE,
     FIRST_FTN        => MINUS_ONE,
     LAST_FTN         => MINUS_TWO,
     K                => INITIAL_VALUE,
     Z                => INTEGER,
     MIDDLE_FTN       => ZERO,
     DEFAULT          => 1,
     BINARY_OP        => "+"                          );
##################################################################
```

Figure 6.1.2

6.2 INSTANTIATING A SIMILAR FUNCTION

Examine the recursively defined function:

f(n) = nf(n-1)/f(n-3) for n > 3 and f(1) = f(2) = f(3) = 3.

We can implement this function in the following recursion:

```
-----------------------------------------

function THREE ( P : POSITIVE ) return FLOAT is
     D : FLOAT := 3.0;
begin -- THREE
     if P > 3 then
          D := (FLOAT(P)*THREE(P-1))/THREE(P-3);
     end if;
     return D;
end THREE;

-----------------------------------------
```

Figure 6.2.1

There is an exact analogy between this function and the abstract recursive generic function. If we write

D := THREE(P-1) / (1.0/FLOAT(P)) / THREE(P-3), we have the necessary analogy. Thus, we may change this function directly to iterative form by the following instantiation:

```
##################################################################

function INITIAL_VALUE ( P : POSITIVE ) return BOOLEAN is
begin
     return N in 1..3;
end INITIAL_VALUE;

function MINUS_ONE ( P : POSITIVE ) return POSITIVE is
begin
     return P-1;
end MINUS_ONE;

function MINUS_THREE ( P : POSITIVE ) return POSITIVE is
begin
     return P-3;
end MINUS_THREE;

function ONE_OVER_P ( P : POSITIVE ) return FLOAT is
begin
     return 1.0/FLOAT(P);
end ONE_OVER_P;

fucntion THREE is new ITERATIVE_ANALOGY (
     T               => POSITIVE,
     FIRST_FTN       => MINUS_ONE,
     LAST_FTN        => MINUS_THREE,
     K               => INITIAL_VALUE,
     Z               => FLOAT,
     MIDDLE_FTN      => ONE_OVER_P,
     DEFAULT         => 3.0,
     BINARY_OP       => "/"                    );

##################################################################
```

Figure 6.2.2

7. ANALOGIES AND LIL

7.1 TWO SIMILAR FUNCTIONS

We examine two recursive functions taken directly from McGettrick (1982):

a) The 3x + 1 Function

F(N) = if N mod 2 = 0 then N/2 else F(F(3*N + 1)) end if

It can be shown that this function terminates for all N in NATURAL (the nonnegative integers). In fact, if N is even, F(N) = N/2; otherwise, setting N (uniquely) to (2 ** h) * k + (2 ** (h-1)) -1 for some h, k in NATURAL, then

F(N) = (3 ** (h-1)) * k + (3 ** (h-1) - 1)/2.

b) McCarthy's 91-Function

F(Z) = if Z > 100 then Z - 10 else F(F(Z + 11)) end if

Again, it can be shown that this function terminates for all integers Z. In fact, if Z is greater than 100, then F(Z) = Z-10; otherwise, F(Z) = 91.

7.2 THE 3x + 1 FUNCTION

Examine first the 3x + 1 function for making abstractions. Notice that this function contains three controlling internal functions: K(N) = (N mod 2 = 0), G(N) = N/2, and H(N) = 3*N + 1. So we can write this function as F(N) = if K(N) then G(N) else F(F(H(N))) end if. It is also true here that K(N) false implies then K(H(N)) true.

So, writing this in Ada, we have

```
-----------------------------------------
function F ( N : NATURAL ) return NATURAL is
     A, B, C, D : NATURAL;
begin -- F
     if K(N) then
          A := G(N);
          return A;
     else
          B := N;
<<FF>>    C := G(H(B));
          if K(C) then
               D := G(C);
               return D;
          else
               B := C; goto FF;
          end if;
     end if;
end F;
```

Or, we can write

```
function F ( N : NATURAL ) return NATURAL is
     M : NATURAL;
begin -- F
     if K(N) then
          return G(N);
     else
          M := G(H( N ));
          if K(M)  then
               return G(M);
          else
               return F(M);
          end if;
     end if;
end F;
-----------------------------------------
```

Figure 7.2

Or, F(N) = if K(N) then return G(N) else F(G(H(N))) end if.

This new definition of the 3x + 1 function is based on the fact that if K(N) is false then K(H(N)) is true. So, we have simplified the function from the composition of two recursive functions to a single function call.

Define a special ordering on NATURAL: Y ORD X if and only if NOT K(X) and there is a sequence X0 = X, X1, X2,...,Xn = Y in NATURAL such that Xi+1 = G(H(Xi)) for i = 1, 2,...,n-1. Thus, it can be shown that NATURAL is a well-founded set with ordering ORD as described in Figures 4.1.2 and 4.1.3 (see the Appendix A for a proof).

Therefore, we have made K( ) an END_OF_CHAIN_FUNCTION and G(H( )) a REDUCTION_FUNCTION, and termination of the function F is assured.

7.3 STRUCTURING ADA TO USE LIL

To build the LIL generic package notice that we already have all the basic theories: WELL_FOUNDED_SET, REDUCTION_FUNCTION, SINGLE_VARIABLE_FUNCTION and END_OF_-CHAIN_FUNCTION. We must code the following to utilize LIL specifications effectively:

```
-----------------------------------------

function THREE_X_PLUS_ONE ( N : NATURAL )
                return NATURAL is
begin
     return 3*N + 1;
end THREE_X_PLUS_ONE;

function X_OVER_TWO ( N : NATURAL )
                return NATURAL is
begin
     return N/2;
end X_OVER_TWO;

function X_EVEN ( N : NATURAL )
                return BOOLEAN is
begin
     return ( N mod 2 = 0);
end X_EVEN;

function X_LT ( N1, N2 : NATURAL )
                return BOOLEAN is
begin
     if ((X_EVEN(N1) and X_EVEN(N2)) or
                (N1 < N2)) then
          return FALSE;
     elsif (N1 = X_OVER_TWO(THREE_X_PLUS_ONE(N2)))
                then
          return TRUE;
     else
          return  X_LT( N1,
            X_OVER_TWO(THREE_X_PLUS_ONE(N2)) );
     end if;
end X_LT;

-----------------------------------------
```

Figure 7.3

7.4 BUILDING THE LIL PACKAGE

```
=========================================

generic package COMPOSITION_RECURSIVE_FUNCTION [
     ELT        :: WELL_FOUNDED_SET;
     REDUCE     :: REDUCTION_FUNCTION[ ELT ];
     EOC        :: END_OF_CHAIN_FUNCTION[ ELT ];
     FTN        :: SINGLE_VARIABLE_FUNCTION
                          [ ELT; ELT]
                                              ] is
function
     COMP       : ELT -> ELT;
vars
     INPUT      : ELT;
axioms

     ( COMP(INPUT)= if EOC(INPUT) then FTN(INPUT);
                    else COMP(FTN(REDUCE(INPUT))) );

     ( if notEOC(INPUT) then EOC(REDUCE(INPUT)) );

     ( FTN(REDUCE) :: REDUCTION_FUNCTION[ ELT ] );

end COMPOSITON_RECURSIVE_FUNCTION;

=============================================

view NATURAL_LT :: WELL_FOUNDED_SET => NATURAL is
     types ( ELT => NATURAL )
     ops   ( ORD => X_LT )
end NATURAL_LT;

=============================================

view THREEXPLUSONE :: REDUCTION_FUNCTION =>
                      THREE_X_PLUS_ONE is
     types ( ELT => NATURAL_LT )
     ops   ( REDUCE => THREE_X_PLUS_ONE )
end THREEXPLUSONE;

=============================================

view EVEN :: END_OF_CHAIN_FUNCTION => X_EVEN is
     types ( ELT => NATURAL_LT )
     ops   ( EOC => X_EVEN )
end EVEN;

=============================================
```

```
view HALF :: SINGLE_VARIABLE_FUNCTION =>
                        X_OVER_TWO is
     types ( ELT => NATURAL_LT )
     ops   ( FIN => X_OVER_TWO )
end HALF;
```

```
make THREE_X_PLUS_ONE_FUNCTION is
    COMPOSITION_RECURSIVE_FUNCTION
         [NATURAL_LT; THREEXPLUSONE;
              EVEN; HALF] end
```

Figure 7.4

7.5 THE ANALOGOUS PROBLEM

Notice now that by examining McCarthy's 91-function and the generic package COMPOSITION_RECURSIVE_FUNCTION we find several analogies. However, there is a problem! McCarthy's function can be written as

F(Z) = if Z > 100 then Z - 10 else F(F(Z + 11)) end if.

If Z < 90, then Z + 11 is NOT > 100; however (Z + 11) > 100 is REQUIRED by the specifications of the LIL generic package, COMPOSITION_RECURSIVE_FUNCTION. We could change the package spacifications or simply alter the domain of McCarthy's function to 90, 91,.... If we allow all of the Ada type INTEGER in the domain, McCarthy's function still terminates. Therefore, with minor adjustments, we have made an exact analogy. We can now build the corresponding Ada and LIL structures to instantiate this function in the LIL environment.

```
-----------------------------------------

type NINETY is range 90..INTEGER'LAST;

function Z_PLUS_ELEVEN ( Z : NINETY )
                  return NINETY is
begin
     return Z + 11;
end Z_PLUS_ELEVEN;

function Z_MINUS_TEN ( Z : NINETY )
                  return NINETY is
begin
     return Z - 10;
end Z_MINUS_TEN;
```

```
function Z_LARGER_THAN_100 ( Z : NINETY )
                return BOOLEAN is
begin
     return Z > 100;
end Z_LARGER_THAN_100;

function Z_LT ( Z1, Z2 : NINETY )
                return BOOLEAN is
begin
     return (Z2 < Z1) and (Z1 <= 101);
end Z_LT;
```

Figure 7.5.1

```
view NINETY_LT :: WELL_FOUNDED_SET =>
                  NINETY is
     types ( ELT => NINETY )
     ops   ( ORD => Z_LT )
end NINETY_LT;

view ELEVEN :: REDUCTION_FUNCTION =>
                  Z_PLUS_ELEVEN is
     types ( ELT => NINETY_LT )
     ops   ( REDUCE => Z_PLUS_ELEVEN )
end ELEVEN;

view LARGER :: END_OF_CHAIN_FUNCTION =>
                  Z_LARGER_THAN_100 is
     types ( ELT => NINETY_LT )
     ops   ( EOC => Z_LARGER_THAN_100 )
end LARGER;

view TEN :: SINGLE_VARIABLE_FUNCTION =>
                  Z_MINUS_TEN is
     types ( ELT => NINETY_LT )
     ops   ( FTN => Z_MINUS_TEN )
end TEN;

make MCCARTHYS_91_FUNCTION is
     COMPOSITION_RECURSIVE_FUNCTION
          [NINETY_LT; ELEVEN;
               LARGER; TEN] end
```

Figure 7.5.2

7.6 A LIL TO ADA TRANSFORMATION

Producing an Ada generic function now is simple. All the necessary information is built into the LIL generic package COMPOSITION_RECURSIVE_FUNCTION of Figure 7.4.

```
-----------------------------------------

generic
     type ELT is private;
     with function REDUCE( E : ELT ) return ELT;
     with function EOC( E : ELT ) return BOOLEAN;
     with function FIN( E : ELT ) return ELT;
function COMP ( INPUT : ELT ) return ELT;

-----------------------------------------

function COMP ( INPUT : ELT ) return ELT is
begin -- COMP
     if EOC(INPUT) then
          return FIN(INPUT);
     else
          return COMP(FIN(REDUCE(INPUT)));
     end if;
end COMP;

-----------------------------------------
```

Figure 7.6.1

We also can have an iterative version of COMP called ITERATIVE_COMP with exactly the same generic specification except for the name, of course.

```
-----------------------------------------

function ITERATIVE_COMP ( INPUT : ELT )
                                    return ELT is
     RET : ELT := INPUT;
begin -- ITERATIVE_COMP
     while not EOC(RET) loop
          RET := FIN(REDUCE(RET));
     end loop;
     return FIN(RET);
end ITERATIVE_COMP;

-----------------------------------------
```

Figure 7.6.2

Of course we can choose this Ada generic implementation by using LIL. This should be done for purposes of run-time effeciency, etc.

```
make THREE_X_PLUS_ONE_FUNCTION is
     COMPOSITION_RECURSIVE_FUNCTION
          [NATURAL_LT; THREEEXPLUSONE;
               EVEN; HALF] =>
                    ITERATIVE_COMP end
                    --------------
-- and so forth.
```

Figure 7.6.3

We can now map this LIL instantiation to an Ada instantiation by Figure 7.6.4.

```
Function THREE_X_PLUS_ONE_FUNCTION is new
     ITERATIVE_COMP (
          ELT       => NATURAL,
          REDUCE    => THREE_X_PLUS_ONE,
          EOC       => X_EVEN,
          FTN       => X_OVER_TWO
                    );
```

Figure 7.6.4

8. LIL IN AN ADA ENVIRONMENT

Burstall and Goguen (1981) and Goguen (1984) describe attempts to work only with specification languages that are divorced from the implementation environment, that are concerned with only abstract higher-order logic program design, or that are designed especially to be manipulated by a syntax checker. Such languages can be very useful in requirements engineering environments to verify specifications; however, there is not yet an automated transition from program specifications to program implementations.

Placing a specification language into a program development environment where "ordinary" programmers labor could be a fruitless effort - especially in making reusable code. The language should support some degree of informality for readability while maintaining strict adherence to good logic. It should be able to use some of the Ada tools for its own support, and Ada code should refer to the specifications when links exits between these languages (Litvichouk and Matsumoto 1984).

These specifications should also be able to access the predefined Ada language library units and packages like STANDARD, TEXT_IO, SYSTEM, etc. and any other units that are peculiar to the implementation like MATH_FUNCTIONS, SORT_ALGORITHMS, etc. If the later units are utilized, the

language should clearly record this act with a "use" statement; doing this will support a degree of implementation independence of the language.

LIL meets these criteria. Although we have not demonstrated all the possibilities of LIL expressed by Goguen (1986) like hide types, hide ops, initialization, and hidden implementations, these additional qualities add much weight to the suggestion that LIL could effectively be placed in an Ada development system.

Library management of LIL and Ada code may actually be quite complex. Some LIL code may have to be made to be readable (as text or graphs) by humans, while other files may only be machine readable for the purposes of syntax checking, linking with other LIL files, enforcement of LIL to Ada transformations, and data base management.

As an example, if we examine the demonstration of LIL and Ada in Chapter 7, we see that after making a general analogous abstraction we were able to move well "above" an Ada generic specification to the LIL generic package COMPOSITION_RECURSIVE_FUNCTION. However, the theories in the requirements theory had to be predefined in LIL. This could have been done even before the addition of LIL in the Ada environment, although some theories like PARTIALLY_-ORDERED_SET and even GROUP can have different but equivalent definitions. Furthermore, even words used in the axioms of these theories like the BOOLEAN "not", "sequence", and "chain" may be defined at an even more abstract level of LIL

theories.

LIL packages themselves are designed to support different versions of some abstract data types. These, of course, are not dependent on Ada code at all - except for predefined library units. Views, on the other hand, may link LIL units to LIL units or LIL units to Ada units that should have been previously written. The latter views could have a notation refering to the specific Ada compilation unit containing the Ada units listed in the views. For example view HALF in Figure 7.4 could contain the information " -- with ADA package THREE_X_FUNCTIONS " since this Ada package contains the function X_OVER_TWO utilized in this view.

A LIL instantiation like "make MCCARTHYS_..." of Figure 7.5.2 that contains no reference to a particular generic implementation has no calls to non-predefined Ada units. Therefore, once the necessary theories have been constructed, the views will link the details of LIL theories with implementable non-abstract Ada.

So, we can think of the major LIL-Ada link as a transformation

T : < LIL generic package, LIL instantiation >

--> < Ada generic package, Ada instantiation >,

where the details of this transformation are set by the views and the implementable non-abstract Ada in these views.

The Ada generic units and instantiation could be annotated to link them with their corresponding LIL units.

For example in the generic specification of the function COMP in Figure 7.6.1 we could have written " -- use LIL generic package COMPOSITION_RECURSIVE_FUNCTION". In the Ada instantiation THREE_X_PLUS_ONE_FUNCTION in Figure 7.6.4 we could have written " -- use LIL make THREE_X_PLUS_ONE_FUCN-TION."

The axioms imported from the LIL generic package by the transformation T, above, would aid the programmer in coding various generic implementations which could be linked later with alternate LIL "make" entities as in Figure 7.6.3.

9. CONCLUSION

Much of the methodology in making analogous abstract generic program units is straightforward and deterministic; however, judgements still have to be made. Since these generic units literally implement abstractions, the notion of a permanent, reusable template (DOD 1983) demands extreme care in making program verifications. Ada, with its separate compilation and with its separation of generic specifications and implementations, supports many modern software engineering standards (Booch 1983), but there is a lack of support in the use of and refinement of abstractions.

In order to utilize such methodologies as analogy programming effectively the programmer must see the semantics of abstract analogies. These semantics are most often expressed, and thus hidden, in the implementation parts of generic units which may not be available to the programmer nor included in a library specified for reuse of code.

LIL opens up these specifications to the entire environment making them not only visible but reusable and standard. Making specifications in LIL is not, as yet, an exact science. However, some of the major goals set forth in our paper (Harrison and Liu 1986) have been satisfied

especially those demanding support for analogy programming and the development of abstract notions within an Ada environment.

LIL is certainly not yet a complete specification language that could be used effectively in an Ada language development system. However, the need to reduce costs and improve reliability in embedded software systems written in Ada calls for techniques which Ada does not completely support on its own.

Many questions still must be answered before LIL or a similar language is fully implementable. How can LIL be accessed effectively by the programmer? How can we use LIL to make specifications for dynamic allocation of memory (access types)? Can LIL be used to specify the actions of Ada tasks? Can LIL be placed effectively in a distributive processing environment that supports an Ada development system?

We have tried to demonstrate the link between analogy programming and the inclusion of a specification language as an essential part of the Ada program development environment. While analogy programming in Ada promotes the reuse of algorithms by the abstraction of specifications, our extentions of LIL maintain these analogies by "forcing" reliability of instantiated code. If we accept that implementations of abstract generic code should be generally hidden from the user, then the inclusion of LIL semantics in the development environment can aid in creating and

maintaining the libraries of generic program units to insure a high degree of trust in the instantiated versions, thus promoting the reuse of code over reimplementation.

Since most of the current methodologies in promoting the reuse of Ada code are based on documenting the code by descriptive identifiers and annotations (St. Dennis et al, 1986), we feel the Goguen's LIL and our extentions of this language (see Appendix B) would not only reduce the need for these documentations but could enforce by automation the appropriateness and correctness of the code use.

The production of a specification language internal to a software development system plus the techniques of modern programming and new implementable methodologies spurned by the development of Ada should give programmers and developers an edge in dealing with the complexity of the software crisis.

REFERENCES

[1] Booch, Grady, Software Engineering with Ada, Menlo Park, Benjamin/Cummings, 1983.

[2] Burstall, R. M. and Goguen, Joseph A., "An Informal Introduction to Specifications Using Clear," in The Correctness Problem in Computer Science, R. S. Boyer and J. S. Moore, Eds., New York, Academic Press, 1981, 185-213.

[3] Colussi, L, "Recursion as an Effective Step in Program Development," ACM Transactions on Programming Languages and Systems, Vol. 6, No. 1, January 1984, pp. 55-67.

[4] Dershowitz, Nachum, "Program Abstraction and Instantiation," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1985, pp 446-477.

[5] Dijkstra, Edsgar W., A Discipline of Programming, Englewood Cliffs, New Jersey, Prentice-Hall, 1976.

[6] DOD, Reference Manual for the ADA Programming Language, ANSI/MIL-STD-1815A-1983, United States Department of Defense, 1983.

[7] Goguen, Joseph A., "Reusing and Interconnecting Software Components," IEEE Computer, Vol. 19, No. 2, February 1986, pp. 16-28.

[8] Harrison, George C. and Liu, Dar-Biau, "Generic Implementations Via Analogies in the Ada Programming Language," AdaLETTERS, (to appear).

[9] Levy, Leon S., Discrete Structures of Computer Science, New York, John Wiley & Sons, 1980.

[10] Litvichouk, Steven D. and Matsumoto, Allen S. "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specifications," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 544-551.

[11] Luckham, D.C. and von Henke, F. W., "An Overview of Anna, a Specification Language for Ada", IEEE Software, Vol. 2, No. 2, March 1985, pp. 9-22.

[12] McGettirck, Andrew D., Program Verification Using Ada, Cambridge, Cambridge University Press, 1982.

[13] Matsumoto, Yoshihiro, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 502-513.

[14] St. Dennis, R., Stachour, P., Frankowski, E., and Onuegbe, E., "Measurable Characteristics of Reusable Ada Software," AdaLETTERS, Vol. VI, No. 2, March, April 1986, pp. 41-50.

APPENDIX B

A WELL-FOUNDED SET PROOF

(see Section 7.2)

Definition: If X and Y are in NATURAL, then define Y ORD X if and only if NOT K(X) and there is a sequence X0 = X, X1, X2,...,Xn = Y in NATURAL such that Xi+1 = G(H(Xi)), where for i = 1, 2,...,n-1.

Theorem: NATURAL with order relation ORD is a well-founded set.

Proof: a) If it were true that X ORD X for some X in NATURAL, then there would be a sequence X = X0, X1,..., Xn = X such that Xi+1 = G(H(Xi)) for i = 1,2,...,n-1. So, Xi+1 = G(H(Xi)) = (3Xi + 1)/2, and Xi < Xi+1. Therefore, by transitive property of "<" in NATURAL, X < X which is a contradiction.

b) Suppose both Y ORD X and X ORD Y for some X, Y in NATURAL. Then there are sequences X0 = X, X1,...,Xn = Y and Y0 = Y, Y1,...,Ym = X such that Xi+1 = G(H(Xi)) and Yi+1 = G(H(Yi)). Linking the two sequences we would have X ORD X, which is a contradiction. Thus, if Y ORD X then NOT(X ORD Y).

c) Transitivity is derived by an arguement similar to b).

d) Suppose A1, A2,...,An,... is a sequence of elements in NATURAL such that Ai+1 ORD Ai. So, by the definition of ORD, A1 is odd. Suppose h > 0 is the position of the first zero on the right in the binary expansion of A1;

thus A1 = (2 ** (h+1))*k + (2 ** h) -1 for some unique k >= 0. Since, ORD is determined by successive applications of G(H(x)) = (3x+1)/2, h applications of G(H( )) leads to (3 ** h)*2*k + (3 ** h) -1, which is even. Since the element on the right of ORD must be odd, the sequence above terminates. Thus all such "decreasing" sequences are finite.

APPENDIX B

OUR VERSION OF LIL

To Goguen (1986), LIL is a tool for automated support of the reusing and connecting of software components and to provide a non-visible, internal tool in a general program development environment. Our goal is to apply this language to an Ada Language System. In doing so we tried to make the language closer in syntatic style to Ada, remove a few syntatic inconsistancies in Goguen's paper, and use more descriptive identifiers in our examples. It is not clear who would build and maintain Goguen's version of LIL in a programming environment. It is our intention to make the programmer responsible for maintenance; thus, this is our justification for using a more Ada-like syntax that still is formal enough to support graphical and natural language interfaces.

Goguen's LIL is used to make and support links among software components without making reference to the Ada language. Our version of LIL intimately and visually links LIL theories with concrete Ada code. In our design Ada generic specifications and initial versions of implementations can be written directly from LIL generic packages. Like Goguen's LIL, "makes" can be produced by using direct "views" from LIL to LIL and from LIL to standard data types. Unlike Goguen, we

use "views" to link LIL theories with Ada subprograms, thus making the creation of Ada instantiations a direct consequence of the production of LIL "makes."

Therefore, the Ada version of LIL involves the programmer in maintaining and using the language so that the Ada environment becomes literally a two-language system with Ada code as the deliverable product while LIL is being enlarged and perfected within the system.