Computer Science Department

TECHNICAL REPORT

RECURSIVE DATA TYPES IN SETL: AUTOMATIC DETERMINATION, DATA LANGUAGE DESCRIPTION, AND EFFICIENT IMPLEMENTATION

By
Gerald Weiss

Technical Report #201
March 1986

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

RECURSIVE DATA TYPES IN SETL: AUTOMATIC DETERMINATION, DATA LANGUAGE DESCRIPTION, AND EFFICIENT IMPLEMENTATION

By
Gerald Weiss

Technical Report #201
March 1986

Recursive Data Types in SETL: Automatic Determination, Data Language Description, and Efficient Implementation

Gerald Weiss

October 1985

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the Courant Institute of New York University.

Work supported in part by the Office of Naval Research, Grant NO0014856K0413.

Table of Contents

ABSTRACT

Very high level languages are often weakly typed in the sense that different occurrences of a name can be associated with distinct types. The types of many entities are nevertheless determinable from the structure of the program, allowing translators for these languages often to incorporate some sort of typefinding algorithm. Due to problems of algorithmic termination, however, these algorithms have been unable to type structures of a recursive nature such as trees. In this thesis we present a method which detects and uncovers the structure of recursive objects, and discuss possible applications of the method to optimization of code. We examine the run-time type model of SETL and the corresponding data representation sublanguage (DRSL), and present a general critique of the design as well as implementation of the current data representation sublanguage. The objects expressible by the latter are shown to be proper subsets of the universe of types assumable by SETL entities at run-time; we present suggestions for extending the representation sublanguage to allow for complete type specification.

Acknowledgements

I would like to express my sincere and deep gratitude to my thesis advisor, Professor Ed Schonberg for his support and guidance. His counseling and advice repeatedly proved to be invaluable to the development of this thesis. I would also like to thank the members of the NYU ADA/ED project, in particular Brian Siritzky and Matthew Smosna, for their help and the many fruitful discussions I have had with them.

I am grateful to my parents for instilling in me the desire to pursue an academic career. Finally, I would like to thank my wife, Fern, and children, Yocheved and Zvi, for their patience and encouragement during the development of this thesis. Without their support I could never have accomplished this.

CHAPTER 1

Introduction

1. Introduction

The semantic level of a programming language is characterized by the degree to which a programmer must pay attention to implementation details: the higher level the language, the less need be supplied and the more natural the expression of the algorithm. In order to facilitate the natural expression of algorithms, many high level languages free the programmer from the burden of statically typing program variables through declarative statements. In addition, data types far removed from those actually implementable via the hardware are provided, again to mask questions of implementation. Thus, compare the classical concordance program written in PL/I, where table handling is the programmer's responsibility, and in SNOBOL where the table is a language-defined data type.

SETL is a set-oriented language developed and implemented at New York University. It is weakly typed and declaration free, and most of its operators are overloaded. As a consequence, there is a substantial overhead in run-time type checking, and only interpretive code can be profitably generated by the SETL translator. To remove the burden of run-time type-checking, and allow the translator to generate efficient machine code, a typefinding algorithm is needed to infer the types of program entities from their use. The static information that the typefinder yields can then be used by a data-structuring module to find the most efficient internal data structures with which to represent composite data objects (sets, tuples and maps). This thesis examines in detail the typing issues present in SETL and proposes substantial extensions to existing type-finding algorithms and to the current type structure.

1.1. The Type Model of SETL

The type model of a language is one of the central issues of the design of the language. Depending upon the type structure, bindings between objects and the types allowable by the language may be possible at translation time or may have to be postponed until run-time. A **weakly typed** language is defined as one in which an object may assume different types during the course of its lifetime. A **strongly typed** language, on the other hand, disallows such freedom, requiring objects to have a single type. This type may be the union of two or more simpler types but the strong type model requires a controlled mechanism (e.g. the discriminant in an Ada variant record) for determining which of the possibilities is currently in force . This notion of strong vs. weak typing is independent of whether or not the language is declaration-free. The programming language B [Meer85], for example, which is strongly typed is nevertheless declaration-free. The distinction between strong and weak typing closely reflects the contrast between mainstream languages (e.g. Pascal, Ada and PL/I) and so-called *very-high-level languages.*

SETL allows the components of data aggregates (i.e. sets, tuples and maps) to themselves be aggregates. This nesting, or embedding, can be extended to an arbitrary depth. Due to this facility, there is a rich variety of types that can be assumed by entities in a SETL program. The introduction of such structures into a strongly typed language is problematic in that the shape and structure of such entities is most often unknown until run-time and can therefore not be typechecked. Furthermore, the set of possible shapes such structures can assume may be infinite posing serious problems in declaring them. There is no problem if pointers exist in the language but SETL (and APL) have no such notion. Even APL, which is weakly typed, is unable to incorporate such objects into the language because it restricts aggregates to be rectangular arbitrarily dimensioned arrays with scalar component types rather than allowing arrays of arrays. SETL itself has had difficulties with such structures in that the automatic typefinder incorporated into the optimizer is unable to assign them any form of meaningful type, and the representation sublanguage, which provides the programmer with the facility to declare names in the program contains no facility for declaring such entities.

1.2. Recursive Data Types

This section focuses upon the class of data types that are typically given a recursive definition. We examine two basic methods of viewing such structures in various programming languages.

The common definition of a recursive data structure, RD, is one whose components are homologous to RD. The term recursive often has another meaning within the context of data structures. Given a linked list, we often say that it is recursive if there is a cycle within the link structure of the list (e.g. a circular list). This definition is of no interest to us and as such we denote all lists, circular or not, as recursive.

The user's view of a recursive structure within the context of a particular programming language is biased by whether the language is pointer- or value-oriented. In a pointer language (such as PL/I or Ada), composite structures whose structure vary dynamically are not directly supported, but rather are built up of simple structures linked together by pointers. Those value languages (languages in which objects are not shared, but must rather be copied) which allow aggregates to be components of other aggregates, on the other hand, allow for dynamic objects of arbitrary length and depth, and thus recursive structures are representable in a more direct fashion. The method of programming in the above two environments is also affected by this distinction. Value based languages are functional in approach, objects constructed via calls to functions; while pointer based languages are more dependent upon the side effects of the assignment statement and parameter modifications. Before discussing the situation in SETL we present three languages: Ada and PL/I which, though both are pointer based, have totally different philosophies concerning the use of pointers in a high-level programming language; and pure LISP which is value based.

1.2.1. Pointers

Pointers have two general areas of application: they can be used to achieve a form of aliasing or overlaying of two variables (most notably of different data types); and they are also used to maintain references to dynamically allocated objects. The primary discomfort expressed by language designers with respect to pointers is in the first area of application. Overlaying allows the programmer to bypass any type protection provided. Pointers as place-holders to dynamic objects do have the problem of dangling references, but few would argue to eliminate dynamic objects on these grounds.

Efforts have been made to allow the use of pointers for dynamic variables while at the same time prohibiting their use in overlaying variables of different types. When used for overlaying, pointers are viewed as addresses and thus may be assigned any value within the range of the address space. No thought is given to the type structure of the program. When it allows pointers to be manipulated in such a fashion, the language normally provides some function (e.g. ADDR in PL/I) that accepts a variable as its input and returns the underlying address of that variables as its result. Note that at this point, there are now two ways of referencing the variable: by its static (or declared identifier), and secondly, via its internal address. The absence of such a function in a programming language prevents the programmer from mixing static and dynamic references.

When pointers are to be used to maintain references to dynamically created objects, their function becomes one of a higher level than that of machine addresses. When a value is created at run-time, it is nameless in the sense that it has no identifier associated with it and thus, the reference to it returned by the allocation function must be saved to allow access to this new object. Whether the reference to the object is an address or something else entirely is of no interest, what does concern us is that it is uniquely associated with the object just allocated.

1.2.2. Recursive Structures in PL/I

Of the pointer-oriented languages we shall examine, PL/I [PL/I76] has the loosest type structure with regard to pointer references. Its pointers are said to be **untyped**, i.e. pointers are merely viewed as machine addresses without any regard to the logical type structure imposed by the declarations upon the corresponding referenced objects. Pointers are simply declared as such and can reference any type of entity in the language. One can additionally declare a variable as BASED on some pointer, with the effect that that variable is then an alias for any object the pointer is currently referencing. Variable overlaying is then easily accomplished as in:

```
DECLARE X     POINTER,
        Y     CHARACTER (4),
        Z     FIXED BINARY BASED (X);

X      = ADDR(Y);
X -> Z = 15;
PUT LIST (Y);
```

The above is a violation of whatever type restrictions PL/I does possess in that it allows the assignment of an integer into an object declared as a character string. One saving grace in favor of PL/I is that it does prohibit arbitrary arithmetic to be directly performed upon pointers.

It is clear from the above that there is no means by which the type of a referenced variable can be determined in PL/I without additional software support on the part of the applications programmer

One advantage of this scheme is to allow the construction of heterogeneous linked structures but at the cost of a total inability to perform any type-checking on pointer-manipulated variables. A related problem, alluded to above, is the inability of the programmer to determine what form of object the current pointer being manipulated is referencing. As a result, what one often encounters in programs dealing with such structures is a LISP-like system, with separate list and atomic nodes, the list nodes being identical in format and containing type descriptors for the atomic nodes they point to.

1.2.3. Recursive Structures in Ada

Ada [Ada83], following PASCAL [Wirt71], is strongly typed with respect to its pointers (called **access types** in Ada). When declaring an access object, the type of object being accessed must also be specified. Pointers are in this manner partitioned as to their use: variables used to reference one type of object may not be used to reference another type. In order to achieve the effect of a nonhomogeneous structure, variant records must be employed. Thus, Pascal and Ada have transformed the pointer into the higher level object discussed above, one whose purpose is to reference dynamically allocated objects. It is for this reason that the designers have chosen to call such an entity an **access type**, rather than using the term **pointer** which has a machine address connotation associated with it. As an example, we present the declaration of a recursive type in Ada [Ada83 pg 3-41]. A record type, CELL, is declared, to represent a typical node of a doubly linked list. An access type, LINK, is also declared, which can point to objects of type CELL:

```
type CELL;
type LINK is access CELL;

type CELL is
  record
        VALUE : INTEGER;
        SUCC  : LINK;
        PRED  : LINK;
end record;
```

The first declaration of CELL is known as an incomplete type declaration and allows the declaration of LINK to specify the type of object it may point to.

1.2.4. Recursive Structures in LISP

In both of the above languages, the programmer, in order to construct and manipulate a recursive structure, must deal directly with the underlying representation, i.e. it becomes his responsibility to manipulate the pointers linking the structure together. It is far more desirable for the programmer to ignore such details and instead concentrate upon the abstract properties of the structure used. Pure LISP [McCa65] allows the programmer such facility. A list is viewed as a sequence of items, some of them lists themselves. High-level operations are provided to allow the programmer to manipulate such lists at their conceptual level, e.g. to insert an element into the list, concatenate two lists together, etc. The fact that internally such structures are maintained in linked format for efficiency reasons is of no concern to the programmer.

The primitive data structure in LISP is the linked list, itself a recursive structure. Trees and more general structures can be easily realized by allowing the elements of a list to be themselves lists. Such constructs are viewed in LISP as being single entities, rather than the linked structures of their counterparts in pointer languages. Indeed, the list structure of LISP is maintained as a binary tree with head (CAR) and tail (CDR) selectors. Such structures vary their shape and size dynamically allowing for a convenient and flexible manner in which to program recursive data structures.

In addition, pure LISP is value-oriented, i.e. objects are never shared. This implies that whenever modification to one structure might affect the value of another, a copy needs to be made to ensure the system's integrity.

In such a system, the problems associated with programming recursive structures in pointer-oriented languages disappear [Hoar75]. Allocation and deallocation of storage is no longer the concern of the programmer, but rather becomes the responsibility of the language processor. Side effects to other structures are no longer possible. Dangling references (i.e. pointers to deallocated storage areas) cannot occur.

If pure LISP is extended with the functions SET, REPLACA and REPLACD, which allow explicit pointer modification of the CAR and CDR values of a list, LISP loses its functional value. Using these functions, it is possible to make a change in one structure and in doing so change the values of other structures. Using these *selective updating* operations, circular structures can be created, and true object sharing can be achieved introducing side-effects and argument modification. It is for this reason that REPLACA and REPLACD are normally used for special-purpose programming only (e.g. garbage collection procedures).

1.2.5. Recursive Structures in SETL

SETL allows for both implementations of recursive structures: pointer and value oriented. While essentially a value-semantics language, the flavor of pointers can be gotten through use of the **atom** data type as in the abstract, pointers establish an explicit mapping between a tag and some corresponding value. In this role whether the pointer is implemented as a physical address or not is irrelevant. The **atom** data type of SETL is akin to the LISP **gensym**, in that each invocation of its generator (**newat** in SETL) yields a name distinct from all others in the program. These can then be used as domain elements of maps to provide the same effect as that of pointers. As an example, a binary tree can be represented in SETL, using the data representation sublanguage by the following three maps:

```
LEFT : map (ATOM) ATOM;
INFO : map (ATOM) INFO_TYPE;
RIGHT: map (ATOM) ATOM;
```

where **map (d) r** denotes a map with domain d and range r. LEFT and RIGHT in this example play the roles normally assumed by pointers.

In addition, SETL also allows for the high-level form of recursive structure representation via arbitrarily nested tuples. Using this approach, each node of a binary tree, for example, can be represented by a tuple, of length 3, whose first value is the left subtree, itself represented in the above fashion; the second element being the information contained in this node, and the last element representing the right subtree:

```
tree: tuple(tree, SOME_TYPE, tree)
```

SETL, however, does not provide any facility for selective updating, i.e. partial modification of a structure, in the manner that LISP allows with REPLACA and REPLACD.

Thus, the statement:

```
x(1) := x;
```

has the same effect as:

```
x := [x, x(2), x(3), ..., x(#x)]
```

and thus no circular structures or side-effects can occur when programming in this manner.

The above two representations are characteristic of two diametrically opposed ways of programming in SETL. The second, using nested tuples, reminiscent of LISP, takes a functional approach, in which new copies of the structure are created for any modifications that are made to the structure. A typical algorithm traversing such a structure will extract components of the structure when traversing it, and upon unwinding the recursion reforms these components into new structures. As an example, consider the following fragment that inserts an element x into a binary search tree:

```
procedure insert (tree, x);

    ....                    $ Handling of leaf cases

[val, left, right] := tree;
if x < val then
  left := insert(left, x);
else
  right := insert(right, x);
end if;
return [val, left, right];
end procedure;
```

in the situation where the node being examined is not a leaf, requiring a traversal further down the tree (i.e. inserting the new element into either the left or right subtree), the three components of the current node are extracted. The appropriate child (left or right) is traversed. Upon returning from the recursive call to insert, a new tuple is formed, consisting of the data element of this node, and the two children, one of them modified.

The first method, employing the maps LEFT and RIGHT as successor functions, is characteristic of programming in a more conventional language, one in which pointers are readily available, the major distinction being the use of logical atoms rather than physical pointers. Modifications are made to the structure in this model, not by creating a new updated copy of it, but rather by making changes to the domains or ranges of the maps. As an example, a typical routine to insert a value into a tree represented by the three maps LEFT, INFO and RIGHT looks like:

```
proc insert(tree, x);

if tree = om then           $ Empty tree
   z := newat;
   INFO(z) := x;
   tree := z;
   return;
elseif x < INFO(tree) then    $ Left insertion
   if LEFT(tree) = om then         $ Leaf node
      z := newat;
      INFO(z) := x;
      LEFT(tree) := z;
      return;
   else                        $ must go down further
      insert(LEFT(tree), x);
   end if LEFT(tree);
else                              $ right insertion
   if RIGHT(tree) = om then       $ Leaf node
      z := newat;
      INFO(z) := x;
      RIGHT(tree) := z;
      return;
   else                           $ must go down further
      insert (RIGHT(tree), x);
   end if RIGHT(tree);
end if x < INFO(tree);
end proc insert;
```

An interesting observation is that SETL is closer to FORTRAN or Algol 60 in this area than to Pascal or PL/I, in that the data structure is 'flattened' into one map per field of all the nodes, rather than a set of records, each containing only the fields pertaining to one node. This is due to SETL's (like FORTRAN and Algol) lack of a record type.

Upon examination of these two modes of programming recursive data structures, it becomes obvious that the pointer-oriented method is the more efficient. This is because less run-time allocation need be performed as successive copies need not be created. If nested tuples are used, a new tuple must be allocated every time a new node is to be added. Additionally, tuples are generated on an extremely temporary basis, producing much garbage, i.e. leading to increased storage reclamation. In the case of maps however, modifying the structure by changing the range value of some item in the domain need not require storage allocation activity.

With regard to programming clarity, elegance, and data structure integrity, the functional approach is preferable. When maps are employed, there is no guarantee of a one-to-one correspondence between nodes in the successor map. For example, given the above binary tree representation, it is possible for the programmer to inadvertently assign the same node as the child of two different parents. This potential hazard is due to the programmer introducing pointers (represented by atoms) into his program, bringing with it the problems of aliasing and side-effects. This cannot occur when programming in a functional value oriented style, as identical children of two distinct parents are maintained as distinct objects.

The above two paragraphs make it clear that we consider it preferable to program recursive structures using nested tuples, and yet want to have the implementation in terms of pointer based structures. Indeed, that is the essence of copy optimization. The typefinding algorithm presented in the first part of this thesis provides us with information that assists us in this change.

1.3. Previous Work

Initial work on typefinding was done by Tenenbaum [Tene74] and his algorithm is the basis of the typefinder used in the current SETL optimizer. In essence, the algorithm passes iteratively over the program deducing type information until no new information is obtained. In the examples presented, the typefinder assigned to 77% of the variables the best type description possible. The remainder were correctly, but overconservatively typed due to 'imprecisions' resulting from the typefinder not exploiting semantic information present in conditional statements.

Jones and Muchnick [Jone76], in presenting the design philosophies of a programming language with late binding times, develop a typefinding algorithm, allowing the programmer to be freed of declaration and data-type restrictions, while at the same time enabling the language translator to implement the program in an efficient manner. Their approach is simpler and more general than Tenenbaum's, though requires more storage, as the resulting system of equations is larger. Kaplan and Ullman [Kapl78] also develop a general method of typefinding and show that their algorithm is yet more comprehensive than Tenebaum's or Jones and Muchnick's.

None of the above algorithms correctly type recursive structures such as trees or linked lists, because the solutions of systems containing structures is infinite and therefore the algorithms may not terminate. Jones and Muchnick [Jone81] subsequently developed a method to discover the type structure of various LISP-like structures. They accomplish this using tree grammars which handles the infinite systems as well as the finite ones. The effect of each statement in a program is regarded as a production in a grammar. Their method, however, does not allow for recursively defined functions, but rather works on a LISP-like language with sequential execution (i.e. the PROG feature of LISP).

Recently, strongly typed languages have been developed which allow the programmer to omit declarations. As static typechecking is mandatory in such a language, typefinding becomes a necessary portion of the translation process and not just an optional part of the optimizer. In addition, typefinding in such a language amounts to *checking* type compatibilities between the various operands of an operation, as opposed to *infering* types of individual name occurrences, as is the case when performing typefinding in a weakly typed language.

ML [Miln83], is a strongly typed language in which most declarations are unnecessary. A typefinding method [Card85] has been developed based upon Robinson's unification algorithm. The method performs tree equivalencing of types, checking types for consistency. Meertens [Meer83] has developed independently a similar method for the programming language B

[Meer85]. With regard to recursive data structures, ML requires explicit declarations of recursive data types, while B disallows structures whose depths vary dynamically.

1.4. Thesis Overview

Chapter 2 of this thesis examines the current state of automatic typefinding in SETL and the inability of the typefinder to strongly type recursive structures that are constructed using nested tuples. An algorithm is presented to remedy this deficiency and its implications discussed. Chapter 3 presents an overview of the data representation sublanguage of SETL, a facility allowing programmer-supplied declarations and aggregate representations. Features currently missing from the sublanguage but which are necessary if recursive structure typing is to be permitted are discussed as well as several other facilities to provide the programmer with the capability to type the objects of the program in a more precise manner. A proposed syntax is suggested to incorporate these changes into the language. Chapter 4 is an implementation, written in SETL, of the typefinder presented in Chapter 2. Chapter 5 presents a variety of example displaying the power of the typefinder, and Chapter 6 presents our conclusions and discusses further research in this area.

CHAPTER 2

A Typefinding Algorithm for Recursive Data Structures

2. A Typefinding Algorithm for Recursive Data Structures

Declaration-free languages are a twofold convenience to the programmer: first, programs are easier to write by not having to include declarations for objects; second, the lack of declarations allows the programmer to defer representation decisions. This latter feature is particularly true for languages in which operators are overloaded. Thus, if the syntax for array access and table (or map) access are identical, the programmer can defer whether to represent a particular structure via a table or array until later in the programming lifecycle. In addition, internal representations for composite structures (such as sets and maps) can be deferred until after the algorithm has been tested.

Unfortunately, there is a price to pay with regard to run-time overhead if declarative information is omitted from a program. Given an overloaded operator, in the absence of any additional information, the code generated for that operator is a call to a run-time library routine which determines the appropriate operation to perform, based upon the actual types of its operands. This run-time decision is unnecessary if the types of objects are determinable at translation time, in which case type-specific operations, possibly implemented using the hardware instructions, can be inserted into the code.

The basic method of eliminating run-time typechecking is to have the translator determine from the structure of the program the possible types a variable can take (assuming that the program is correct). The problem differs for weakly and strongly typed languages in that the typefinder of strongly typed language has the additional knowledge that the same variable must be of the same type throughout the program and thus the task of the typefinder is to perform type consistency checks. Weakly typed languages must also take into account for the possibility of the same variable assuming objects whose types differ at various points in the program. This requirement has an impact upon the design of the language itself. For example, B which is a strongly typed, declaration-free language requires upon input a 'sample' value corresponding to the type of the input variable (e.g. READ n, s EG 0, ''). Without this value, the typefinder would be unable to check for consistent usage of that variable.

2.1. An Informal Introduction to Typefinding in SETL

The basic technique of typefinding is to scan the program examining the manner in which variables are defined and used and to determine from such information the set of possible types assumable by each variable. The information regarding types is obtained in two separate steps: a forward analysis of the program in which type information is propagated from definitions, and a backwards analysis in which types are deduced from the manner in which objects are manipulated.

In the case of SETL, the result of typefinding a variable may be a set of types, rather than a single type. Furthermore, determining a useful subset of the universe of all possible types may not always be possible. In the following program:

```
program p;
read (a);
print (a);
end program;
```

the variable a must of necessity remain untyped.

However, in many instances variables are initialized by means of constant values which betray their types, and defined or used in operations that are type specific (e.g. **x := #t**, where # is the cardinality operator, implies that x is an integer). Furthermore, if structured programming style is employed, the majority, if not all of the variables in a program will be given a single specific type, as the user will employ a variable for one specific purpose only. As mentioned previously, if types can be assigned to such variables, the specific operation can be inserted into the object module as opposed to a call to a general run-time routine.

The presence of weak typing in the type model requires us however, to assume the worst and consider the possibility that a variable is used in an arbitrary fashion and assumes disparate types at various points and times during execution. Thus, given the following:

```
if somecondition then
  x := ...              $ an expression of type t1
else
  x := ...              $ an expression of type t2
end if;
  ...
some use of x             $ having type t1 or t2
```

since the language is weakly typed, rather than checking the two definitions of x for consistency, we are forced to take the union of their types as the potential type of x prior to the subsequent use.

In spite of the seemingly pessimistic system that we have to work with, structured programs will not contain such quirks and the two definitions to x will typically be of the same type, resulting in a strict type even after the union, with the exception of recursive types.

There are also situations in which type errors can be flagged despite the weak typing of the language. Consider:

```
x := 3;
y := arb x;     $ extract an arbitrary element from x
```

arb is a type specific operator used for set extraction. As the only possible type for x from its prior definitions is *integer* prior to performing the **arb**, the typefinder can safely indicate an error.

We noted above that nothing can be said about the type of a variable that is read in. However, if we examine the subsequent uses of that variable, we may infer the set of possible types that variable may assume and still be consistent with its uses. It is still not the case that we can strongly type the instance, but we may move the type check from the point of use back to the point of definition (the **read** in this instance), and in all probability reduce the number of type checks performed, given that the uses being executed are typically more frequent than the definition.

2.2. Terminology

The predefined elementary or primitive data types of SETL are **integers, reals, strings, atoms, and om** (the undefined value). These are combined into more complex objects using set and tuple constructors in conjunction with the special primitives **nullset** and **nulltup** which represent the null set and null tuple (or sequence) respectively. Implicit conversions are not performed.

Given a type, t, a set whose components are of type t is denoted **set(t)**. A tuple of known length with component types t1, t2, ... tn is written as **sequence(t1, t2, ..., tn)**. A tuple of unknown (i.e. arbitrary) length with component type t is written as **tuple(t)**.

When dealing with types, we employ the above notation, rather than denoting a set of component type as **{t}**, and a tuple as **[t1, t2, .. tn]**, to make clear the distinction between the type of an object and the object itself. That is, an object that is a set of integers might be denoted as **{1, 2}**, while its type is denoted by **set(integer)**.

Given types t1 and t2, their **alternation, t1 | t2**, is the type consisting of the union of the sets of domain values valid for t1 and t2. If a type symbol t is expressible into the form **t1 | t2**, then t is said to be an **alternated type**, Otherwise it is called an **elementary type**. t1 and t2 in the above are called the **alternands** of t.

Examples:

set(integer | string) corresponds to a set whose components may be either integers or strings.

sequence(integer, integer) is a pair, both of whose elements are integers.

tuple(real) is a tuple of arbitrary length with real components.

The above notation is insufficient to type exactly many structures. For example, to represent an unknown length tuple whose first element is a string followed by an arbitrary number of integer components, we have no recourse but to write **tuple(string | integer)**. As another example, a binary tree represented via arbitrarily nested sequences, has as its representation an infinite number of alternands. Employing the above notation forces the tree to remain untypeed, resulting in quite a loss of knowledge about this data structure. Note that programming a recursive structure using explicit links (e.g. using atom based maps), can be easily represented as the range and domain of the maps are uniform. It is only for recursive structures represented as dynamically nested tuples that no representation can be given.

For purposes of typefinding, we assume the SETL program to be schematized, i.e. to be transformed into some standardized form convenient for analysis. We employ an extended form of quadruple, where a typical binary operation is represented by an operator followed by an output variable and two input operands. Since SETL is weakly typed, and as such a variable may assume more than one type through its lifetime, we type individual instances of the variable rather than the variable as a whole. The variable's type is subsequently the union of the types of its instances. The appearance of a variable in an instruction is said to be an **occurrence** of that variable. If it is the output of the instruction, it is called an **ovariable** (or **ovar**), otherwise it is said to be an **ivariable (ivar)**. Ovariables are also known as **definitions** and ivariables as **uses**. Given an occurrence of a variable v in statement **i**, we denote that occurrence by **vi**. If there is more than one occurrence of the same variable in the same statement, we number them sequentially from left to right. Instances that are denotations have the denotation bracketed in tildas. For example in the statement:

(4) $x := \{y, 5\} + x$

the occurrences from left to right are $\mathbf{x}_{4.1}$, $\mathbf{y}_4$, $\mathbf{\tilde{}\,5\,\tilde{}}_4$, $\mathbf{x}_{4.2}$.

A definition, v_i, is said to **reach** a use, v_j, of the same variable if there is a path clear of any further definitions to that variable from v_i to v_j.

2.2.1. The BREACHES Chaining

The typefinding algorithm presented in this thesis requires a more general form of value reaching than the one defined above, akin to the value-flow analysis presented in [Schw75a]. Intuitively, we are interested in all occurrences that are dependent, in any manner upon a particular value. If the value in instance v_i reaches instance v_j in this sense, we say v_i **value-reaches** v_j. For example:

```
(1)      x := 5;
(2)      x := {x};
(3)      y := x + {'cat'};
```

the occurrence ˜ 5˜ $_1$ value-reaches y_3 even though it is embedded into a set along the way since y_3's value is in some manner dependent upon the occurrence ˜ 5˜ $_1$.

We now formally define value-reaching. For value-reaching paths of length 1, UD chains are sufficient to define BREACHES from ivariables to ovariables (in fact that is what is used for calculating reaching definitions in optimizers). However, we are interested in extending these links to paths of length greater than 1. We are unable to take the closure of the UD relation as the domain (ivars) differ from the range (ovars). Instead we introduce a new relation, BREACHES, whose domain and range is the set of **all** occurrences.

```
BREACHES(occ) =
    if occ is an ivar then
        UD(occ)
    else
        {ivars of instruction #i} if occ is the
            ovar of instruction #i
```

We are now able to calculate the closure, BREACHES*, which corresponds to going backwards along a path whose links are the UD chains and ovar-ivar relationship.

An instance v_i is now said to **value-reach** instance v_j if $\mathbf{v_i \in BREACHES^*(v_j)}$. We can similarly define the relations FREACHES and FREACHES* that builds its path in a forward fashion. Note the BREACHES* relation is similar to the relation **crpart** in [Schw75a].

The major distinction is that crpart only takes into account component embedding an extraction to propagate values while BREACHES propagates values across all operations.

2.3. Tenenbaum's Typefinding Algorithm

Given the potential program speed-up achievable by moving typechecking from run-time to compile time, the development of a typefinder for SETL programs is an important issue in the implementation and optimization of the language. As mentioned previously, the original algorithm was developed by Tenenbaum, and though improved upon, the basic structure has remained the same since its original description.

A lattice (a partially ordered set with join and meet) of types is presented, which describes a useful subset of the types assumable by SETL variables during execution. This set is similar to the one described in the previous section, with the addition of two primitives which are not actual types in the languages but rather reflect levels of knowledge about the type of a variable. The type symbol **error** is used to denote an erroneous type (either because of an uninitialized value or a usage that is incompatible with the variable's definitions). The type symbol **general** is used to indicate that the type of an instance is unknown. This can arise in two different ways. First, the instance may be the output variable of a **read**, in which case nothing can possibly be known about the type of the instance. Second, the type of a variable may become too complex or require some representation that the lattice structure does not possess and may thus be forced to **general**. The first is unavoidable, we attempt to eliminate occurrence of the latter.

The constructors **set, tuple, sequence** and | are syntactic symbols used to create new lattice elements from already existing ones (as opposed to operators which given an existing element would result in another existing element). The elements of the lattice consist of symbols constructed from the primitives and constructors, with the restrictions that set and tuple constructors can be composed with each other to some maximum nesting, and the length of known length sequences is also limited (the value used in the SETL optimizer for both instances is 6). Levels and sequences beyond that limit are collapsed into **general** and any relevant type information is therefore lost. This is necessary, as is seen below, to ensure termination of the typefinding algorithm.

Unlike many other data flow analysis methods, the typefinder requires the definition of both a meet and a join operation. The join operation, **V**, or **dis** (disjunction), represents the coarsening of type information provided by two type symbols. This corresponds to the merging of two execution paths into a single one. Beyond the point of merging, the type of a variable can be the type of that variable on either of the two paths prior to the merge point. Intuitively, the result of joining two type symbols is no more than the type symbol resulting from the two original symbols separated by the alternation constructor. However, the actual result is often some less precise type, for reasons of efficiency or because no practical use can be made of the more precise information. For example, **dis(tuple(t1),tuple(t2))** collapses into **tuple(t1 | t2)** rather than **tuple(t1) | tuple(t2)**.

The meet operation, **Λ**, or **con** (conjunction), reflects the refining of type symbols into a more precise symbols. One programming situation that calls for the application of this function is in determining the type of a variable from its uses. If several uses of a variable are all unconditionally executed after the same definition of that variable, then the type of the definition can be made more precise by taking the meet (or intersection) of the type symbols of the uses. As an example,

```
(1)    read (x,y);
(2)    w := x + y;
(3)    z := arb x;
```

Forward analysis provides us with little information. x_1 and y_1 are assigned **general** because of the **read**. The instance w_2 can be either **integer, real, string, set or tuple**, which is not much of an improvement over **general**. z_3 must be an integer (by definition of the length (#) operator). Working backwards, however, we see that x_3 must be a set (as that is the input type legal for #), and x and y must be **integer, real, string, set** or **tuple**. As both statements 2 and 3 are executed unconditionally following the **read**, the type of the value read in for x must be a legal one for execution of *both* of those statements. The type of x_1 is therefore the conjunction (or intersection) of the types of x_2 and x_3, resulting in **set**.

Finally, a partial order, $<$, is defined on the lattice as:

$$\mathbf{t1} \overset{\cdot}{<} \mathbf{t2} \equiv \mathbf{dis(t1,\ t2)} = \mathbf{t2}$$

that is, if t2 is a type generalization of t1. Imposing a limit upon the nesting of the set and tuple constructors together with limiting the length of known length sequences guarantees that all chains in the lattice (defined by the above ordering) are finite in length [Tene74 Section 1.2, Theorem 2, and Section 1.6, Theorem 8).

While the meet and join operations are used to reflect the merging and diverging of program flow between the basic blocks of a program, a set of functions are needed for the propagation of type information through the blocks themselves. A pair of such *propagation functions* is defined for each operator: one corresponding to the forward flow of type information from input variables to the output variable of the operation; the other used to determine the type of an input variable given the types of all other operands of the operation.

The typefinder functions in two phases. First, type information is propagated in a forward fashion through the flow graph. During this phase, the first of the above two propagation functions are applied. Constants are assigned their *a priori* types and operands of type specific operations are also assigned types. The information generated by this phase is based upon the manner in which variables are defined in the program. During this phase, only the join operation is employed.

The second phase of the typefinder uses the information of the first phase as well as examining the manner in which variables are used in the program. Type information is propagated in both directions in this phase. This is to allow information obtained from backwards analysis to be propagated forward again resulting in more precise typing. In the preceding example, once the type of x_1 is discovered to be integer, that information should is propagated forward to x_2 and again to w_2, and finally from w_2, backward to y_2. In this phase, variables within input statements can be assigned the set of types they might assume and still be used legally in later statements.

The information derived from the first phase of the typefinder allows the compiler to replace run-time calls to generic library routines with low level, type-specific code sequences if a variable is strictly typed (i.e. its set of types is a singleton other than **general** or **error**). Type information from the second phase allows typechecking to be moved away from the use of a variable back towards its definition (which presumably is executed fewer times), but does not strictly type the variable (i.e. the user may still supply an illegal value for x in the data).

In both phases, the algorithm is iterative and based upon a workpile [Hech77]. As new information is derived, all variable instances which potentially might be affected by such information are placed on the workpile for subsequent processing. The lattice being bounded (all chains are finite due to manner in which the meet and join operations are defined and the limitations upon nesting and sequence length) together with the fact that in the first phase the types of variables are monotonic increasing and in the second monotonic decreasing, guarantees that the workpile will eventually become empty. Due to the nature of the workpile, the phases proceed in a semi-chaotic (i.e non-deterministic) fashion. That is, little can be said about the order in which the variable occurrences are processed.

2.4. Limitations of Tenenbaum's Typefinding Algorithm

As we have seen, Tenenbaum's typefinding algorithm is based upon creating a lattice of type symbols corresponding to the possible types a SETL object might assume at run-time. In order to ensure that the type lattice used in the typefinder be bounded, certain simplifications must be made to the actual type model. The first such change is the limit on the nesting depth of objects. Tenenbaum arbitrarily chose 3 as the maximum depth maintained by the typefinder for any object. Beyond that, compression of levels is performed, the innermost type symbols transformed into **general**. For example, if in the following code fragment the type of t is **set(set(set(integer)))**:

```
s := {t};
```

then the result type (of s) is:

set(set(set(general))),

rather than:

set(set(set(set(integer)))

A second area of simplification is in the disjunction of tuples of differing length. Two tuples whose lengths are known but not identical are merged into a tuple of unknown length with a component type consisting of the disjunction of all component types of the two original tuples:

```
dis(<t1, ..., tj>, <t1', ... tk'>) =
     [dis/[t1, ..., tj, t1', ...tk']]  where j /= k.
```

The first restriction prevents infinite chains of the form:

tg >= {tg} >= {{tg}} ...

while the second guarantees that although there is an infinite number of known length tuple type symbols in the lattice, an infinite number of such symbols cannot appear in a chain.

The above two changes to the actual type model, while necessary to the algorithm for the purposes of termination, result in the loss of information concerning the types of certain objects. Some of these objects, those that continually embed their previous values within their current values and are thus composed of an unbounded number of alternands, are the cause of the above changes and would exceed any nesting depth chosen. Other objects are classified as **general** by virtue of the fact that they exceed the chosen depth even though they can be expressed as a finite number of alternands.

Examples:

a. Given the following code fragment:

```
1   s := 5;
2   (while some_condition)
3     s := {s};
4   end;
```

The occurrence s_1 is placed onto the workpile by the initialization phase since its right hand side is a constant. Calculation of its type as **integer** (via the propagation function for an assignment statement) causes the occurrence $s_{3.2}$ to be placed onto the pile. It is assigned the type **integer** (the disjunction of **error** - occurrence $s_{3.1}$ and **integer** - occurrence s1). Occurrence $s_{3.1}$ is then assigned the type **set(integer)**. Without imposing a nesting depth, the type symbol of s would continually be changing, increasing in the number of its alternands, each iteration producing an alternand which is a set whose component type is the previous iteration's type symbol:

integer | {**integer**} | {{**integer**}} | ....

b. In the following fragment:

s := [**1**, [**2**, [**3**, [**4**, **5**]]]]

(a LISP-like representation of a linked list), the innermost tuple is collapsed to **general** as its depth exceeds the maximum level. It is important to note, however, that s could be typed without fear of nontermination of the algorithm. It is only because of the imposition of an arbitrary limit upon the nesting of objects that the innermost type of s is transformed into **general** (and not due to an infinite number of alternands).

We therefore see that there are two flavors of objects for which information is lost:

a) Objects that would cause the algorithm to loop infinitely producing for them an ever-increasing number of alternands.

b) Objects that 'innocently' trigger the nesting limit but are not intrinsically threatening to the algorithm's termination.

Type symbols which can generate a potentially infinite number of alternands given a finite set of initial types (those of part a), we denote as **nonconvergent types** as they do not converge using Tenenbaum's algorithm. Such objects need to be eliminated from the typefinding mechanism to guarantee convergence. The second category of types (part b), however, are transformed to **general** due to their static complexity It has been noted by [Fong75] that a superset of the set of nonconvergent types can be detected in a program in several ways, most notably by looking for variables with type self-dependencies. Fong, Kaplan and Ullman construct a directed acyclic graph (DAG) of the type dependencies and search for situations in which there is a path in the graph from the type of an instance prior to a loop to the type of the instance within the loop and which passes through a tuple or set former. Such variables produce a new type upon each iteration over the graph, the new type being the previous type embedded in either a set or tuple constructor. Our approach is to look for variable instances that are elements of their own BREACHES* (see next section). Once it is determined which variables are in this class, they can be immediately set to **general**. This can be achieved as a preprocessing pass to the typefinder proper. In such a fashion, arbitrary nesting depths can be eliminated from the typefinding algorithm, as all nonconvergent types will be known to be **general** a priori, and therefore types falling into category b above will not be made **general** because of such limits.

Among the class of nonconvergent types is a subclass which we designate as **recursive data types**, i.e. types containing references to homologous structures. As these variables are nonconvergent, they will be set to **general** by the typefinder, regardless of whether it is supplemented by the above preprocess phase.

A second subclass of the nonconvergent types is the class of **dynamic tuples**, i.e. the set of sequences whose length cannot be determined a priori without executing the program with its input. Tenebaum's typefinder is able to properly type such objects as arbitrary length tuples. However, in doing so, the algorithm also classifies an object that assumes a sequence of length two and length three as an arbitrary length tuple due to the simplification made to the disjunction of disparate length tuples mentioned above. This simplification is necessary in order to prevent code fragments such as:

```
          t := [];
          (while somecondition)
                read (x);
                t := t with x;
    end;
```

from generating an infinite sequence of types of the form:

sequence(general) | sequence(general, general), ...

We thus see that there are two flavors of structure that pose a problem to Tenebaum's typefinder: recursive data structures which are depthwise nonconvergent, and dynamic tuples, whose length is unknown.

2.5. An Accurate Typefinding Algorithm in the Presence of Recursive Data Structures

The primary obstacle to detecting recursive data types using Tenenbaum's typefinder is that the type lattice is infinite and the type of an occurrence which embeds itself will not converge with the standard workpile algorithm. An occurrence **embeds** itself if its value at some point of program execution is later assigned to one of its components. It is this lack of convergence that we overcome with our method, thus allowing recursive types to be uncovered. The new algorithm also distinguishes, in many instances, between dynamic tuples and those that only superficially appear dynamic.

To accomplish this, we first note that the only instances that are candidates for nonconvergence are those that value-reach themselves. The class of occurrences covered by this definition also includes useless cases of the form:

```
(1)   (while somecondition)
(2)      x := x + 1
(3)   end while;
```

since $x_{2.2} \in$ BREACHES*($x_{2.2}$) by definition of BREACHES*. To avoid treating these as recursive (following the method in [Fong75]), we require that the path from an instance to itself contain some form of embedding operator (e.g. tuple former).

We initially detect the set of such self-dependent occurrences and flag them as potentially nonconvergent. This task can be accomplished, in the crudest manner, by examining each and every occurrence in the program and testing it for membership in its own BREACHES*.

Once we have found such an occurrence, we assign it a unique, newly generated type symbol which we designate as a **recursive type symbol**. Associated with this symbol is a type structure, representing the underlying internal structure of the recursive type symbol. When this occurrence is embedded into some larger entity, it is the recursive type symbol, rather than the actual structure of the type that is used. This level of indirection prevents the nonending production of more deeply embedded symbols. To see this, consider the following code fragment:

```
1    t := 3;
2    (while somecondition)
3          t := [t];        $ t embeds itself
4    end while;
```

Traditional typefinding types the ivariable t of statement 3 successively as:

integer, tuple(integer), tuple(tuple(integer)), ...

The recursive structure typefinder, upon discovering that the ivariable $t_{3.2}$ is self-dependent, assigns some new type name, e.g. REC_1, to it. The type structure of REC_1 will then be initialized to **error** much as the type of a nonrecursive occurrence is given the error value as its initial value. We thus distinguish between the type name of a recursive occurrence, which is used for embedding, and the type structure which is employed everywhere else. The tuple forming operation of statement 3 will thus have as its result type

sequence(REC_1)

regardless of the values assigned to the structure of REC_1. When the ovariable $t_{3.1}$ reaches the ivariable $t_{3.2}$, its type structure will have the type **sequence(REC_1)** added to it. However, when the result type across the assignment (i.e. the type of the ovariable $t_{3.2}$) is calculated this second time (and for all future processings of this instruction) the result type will remain **sequence(REC_1)**, and thus the types of both ivar and ovar converge.

Our choice to selectively assign recursive type symbols to ivariables alone results in a simpler set of type equations for the program. The last section of this chapter examines this decision in more detail.

We have shown how the distinction between the symbol of a type and its actual structure eliminates the infinite production of type symbols in the program. However, there are situations when the internal type structure of a recursive variable needs to be examined. Consider:

```
(1)   t := 5;
(2)   (while somecondition)
(3)      t := [t];
(4)   end;
(5)   x := t(1);
(6)   y := x(1);
```

when processing x_5, we get as its type REC_1, and therefore when processing y_6 we must examine the structure of REC_1 too determine the type of its first component so that we can assign it as the type of y_6.

This need to examine the internal structure of a recursive symbol can cause problems in ensuring that type information is propagated to all points of the program. In Tenenbaum's algorithm, whenever new type information is produced, all occurrences which might be immediately affected by the change are marked for further processing. Thus, if a ivariable's type is modified, the ovariable is so marked, as its type is dependent upon that of the ivar. Similarly, if an ovar is assigned a new type, all subsequent uses of that definition must be reprocessed. Changes in the types of these marked instances will mark yet other instances and in this fashion the type information is propagated through the program. To determine which instances to mark, use-definition chains are sufficient.

However when recursive symbols are introduced, UD and DU chains no longer guarantee that new information will be correctly propagated. To see this, consider the above example. The recursive instance is $t_{3,2}$. The operation at that instruction is a tuple-former and thus, if we assign the type symbol REC_1 to $t_{3,2}$, the type of $t_{3,1}$ becomes **sequence(REC_1)** (by definition of the propagation function for tuple-forming). This type never changes regardless of changes to the internal structure of REC_1 and thus $t_{3,1}$ (and therefore t_5, x_5, x_6, and y_6) will never be marked for reprocessing due to a change in the structure of REC_1. Note however, that as discussed above y_6 does have its type affected by changes to (the first component of) REC_1. It is because variable instances at arbitrary points in the program (such as y_6 above) may require information about the structure of recursive symbol that UD chains are no longer enough to guarantee complete type propagation.

This necessity to occasionally examine type structures requires us to revise the manner in which we place new items onto the workpile. As before, when an ovar is processed, all immediate uses of that occurrence are placed on the workpile. Similarly, when an ivar is processed (and a change in its type discovered), the ovariable corresponding to the instruction containing that ivar is placed upon the workpile. While these occurrences are still clearly affected by changes in the processed occurrence, as we have noted above, there are other occurrences that must be placed on the pile (e.g. in the above example, y_6 is affected by changes to the structure of instance $t_{3.2}$), to obtain a complete type propagation.

To solve this problem, we modify the manner in which occurrences are placed into the workpile for further processing. Traditionally, the UD chains determine which occurrences require reexamination. We extend this to include occurrences whose values have to be reexamined because of a change in the type structure of a recursive type. We thus define a new map **occurrences_dependent_upon**, which takes as its domain, the set of recursive type names and as its range the set of occurrences. When the type structure of a recursive type (which is always associated with an ivariable) is modified, those occurrences which belong to occurrences_dependent_upon for that name are placed into the workpile, in addition to the corresponding ovariable. This map is updated with any occurrences that needs to examine the type structure of a recursive type symbol.

When examining the type structure of a recursive type symbol, it is possible that one of its alternands is itself a recursive type name. In such an instance, that symbol's type structure must also be looked at. Fortunately, there are only a finite number of such recursive type names in a program (at most the number of ivariables in the program as recursive symbols are generated only when a self-dependent ivariable is discovered) and we can maintain a stack of all types already examined with respect to the ivariable being processed, and thus we avoid entering any cycles in the type structures. Additionally, we can always omit an appearance of a recursive type name in its own type structure as that is nothing more than an identity. As an example, suppose we have the following recursive symbols:

```
REC_1 : set(REC_2) | REC_3 | integer
REC_2 : string | tuple(REC_1)
REC_3 : real | sequence(REC_3)
```

then the internal structure of REC_1 is:

```
set(REC_2) | integer | real | sequence(REC_3)
```

Note that we need not unfold recursive symbols that are not top-level (e.g. set(REC_2)) as the propagation functions never descend below the component level of a type.

Algorithmically, this examination of the structure of a recursive symbol is as follows:

```
To examine the structure of recursive symbol R:
   - set result to the structure of R
   - while there exists a recursive symbol R' in result:
        remove it from result, place it on a stack,
        merge its structure with result, and remove from
        result any recursive symbols present on the stack
   - result now contains the structure of R
```

The SETL code for the above can be found in procedure **collect_types** in the SETL implementation of the typefinder presented in Chapter 4 of this thesis.

Also note that we never have to go deeper than one level into the structure of a type due to the fact that nested subscripts in the original SETL source are already unrolled at the quadruple level. That is, once the program is decomposed into quadruples, opcodes examine either an operand proper or one of its top level components. Composed subscripts in the original SETL source are transformed into successive subscripting operations using temporary objects to maintain intermediate results.

In the implementation of his algorithm, Tenenbaum states that routines are coded in-line, eliminating the need for interprocedural analysis. However, for purposes of typefinding, treating parameter transmissions as assignments is sufficient. Indeed, for recursive routines, we do not want the types of individual activations of parameters, but rather their overall structure taking into account the recursive nature of the routine. The current SETL optimizer notes this as well, and treats the parameter transmission operations in the same manner as assignments.

The resulting algorithm to typefind recursive data structures is as follows:

Input:

Q1 (quadruple) code of SETL program together with UD chains.

1) The workpile is initialized to any ovariables that have a constant right hand side. A map, **occurrences_dependent_upon**, is initialized to the empty set. The domain of this map is the set of recursive symbols and its range is the set of instances in the program.

2) Whenever an embedding operation (set or tuple formers, **with**, and assignment to a component of a tuple - t(1) := ...) is encountered, the FREACHES* of each ivariable of that operation is calculated. If an ivariable appears in its own FREACHES*, it is flagged as self-dependent.

3) All self-dependent ivariables are given new, unique type names which are flagged as being recursive. A map **type_structure** is created whose domain consists of all such recursive types.

4) The type of all nonrecursive, nonconstant occurrences are set to **error**. The type_structure of all recursive type names is set to **error**.

5) An occurrence is removed from the workpile and processed.

5a) If the occurrence in question is an ovariable, calculate its type using the propagation function appropriate to the opcode of the associated instruction. Furthermore, if the ovariable needs to examine the type_structure of any recursive symbol, place the occurrence in the occurrences_dependent_upon set for that type name. If the type of the ovariable has changed, place into the workpile all subsequent uses of that ovariable.

5b) If the occurrence being processed is an ivariable, calculate its type as the disjunction of all definitions reaching that occurrence. This value becomes the type if the ivariable is nonrecursive, *and the value of the type structure otherwise.* If this value has changed since the last time this occurrence has been processed, place into the workpile the associated ovariable, as well as any occurrences in occurrences_dependent_upon if the ivariable is recursive.

6) Repeat step 4 until the workpile is empty.

In step 5a, examination of the type_structure of a recursive symbol occurs for most operations. It is only in the event of an embedding operation that the structure of a recursive symbol can be ignored. In all other cases, however, the structure must be examined to determine if any of the types within it are legal as input to the operation. For example, if the operation in question is a +, and one of the input operands has as its type the recursive symbol **REC_1**, and the other operand is of type **integer** | **set(real)**, then the structure of **REC_1** must be examined to see whether it can assume **integer** or a set as its type.

2.6. Termination of the Typefinder

Intuitively it is clear that the algorithm should terminate. The operations that cause an infinite number of types to be generated are tuple and set formers, and any other embedding operators (such as **with**). Furthermore, these produce a nonconvergent set of type symbols only in the event that the value of the output variable of the operation reaches one of the input variables. It is precisely for such input variables that a recursive type symbol is introduced (via the self-dependency test). Since it is this name that is propagated across the operation, and not the underlying structure, if the type name reaches one of the defining variables, it will be embedded at at most one level (if there are further tuple formers along the path from the output variable back to the input variable, they will generate other recursive type names that will replace the original recursive name). For example, in the following code fragment:

```
(1)     y := 6;
(2)     (while somecondition)
(3)        x := [y];
(4)        y := {x};
(5)     end while;
```

both y_3 and x_4 are found to be self-dependent and are therefore each assigned its own recursive type symbol, say REC_1 and REC_2 respectively. x_3 is then assigned **sequence(REC_1)** That symbol is propagated to x_4 where it is placed into the type_structure entry for REC_2. y_4 is then assigned **set(REC_2)**. When the operation is again processed, its result type will be identical to the previous result type, namely the recursive type name with whatever transformation is performed by the operation.

Tuple and set forming can no longer expand their result types indefinitely and extraction operation (e.g. **arb**, subscripting) merely introduce types that had been embedded in already existing types and in any event are of a monotonically decreasing nature. Thus, only a finite number of type symbols can be generated by the propagation functions of the program. Since these functions are monotonic, all types must eventually converge. In terms of the generator propagator analogy, what we have accomplished is to eliminate nonconvergent generators from the type equations.

We now present a more formal proof that shows that there is a bound on the nesting depth and length of the type symbols producible by a program. This implies that a program must have a finite number of type symbols, and this fact, together with the monotonicity of the propagation functions guarantees eventual convergence.

We give a precise definition of the nesting depth, **nd**, of a type symbol:

nd(scalar_type) = 0

where scalar_type = **real, int, atom, boolean**, ..., any recursive type symbol, **error** and **general**.

nd(tuple(t)) = nd(set(t)) = nd(t) + 1

nd(sequence(t1, t2, ..., tn)) =
max(nd(t1), nd(t2), ..., nd(tn)) + 1

nd(t1 | t2 | ... | tn) = max(nd(t1), nd(t2), ..., nd(tn))

this function is originally introduced in [Tene74] for the purpose of placing the limit upon a depth of type symbols and thus bounding the lattice.

We similarly define the length of a symbol as:

ln(scalar_type) = ln(set(t)) = -1

ln(NULLTUP) = 0

ln(tuple(t)) = 0

ln(seq(t1, ..., tn)) = n

We show that the nesting depth of any type symbol in a program as well as its length is bounded by the length of the program (in Q1 form). We accomplish this by a case by case analysis of the forms of occurrences that can appear in a program. The analysis concerns itself with ivariables only - if they can be shown to be bounded, ovariables follow immediately (the nesting depth of an ovariable can be at most one more than the maximum nesting depth of its inputs, while the length can be at most the sum of the operand lengths).

Recall that an occurrence is said to be **self-dependent** if it lies on its own BREACHES* path. The occurrences of a program can be divided into two classes: those that are self-dependent and those that aren't. These two categories can be further subdivided in the following manner:

I. Self-dependent

A. Operand of an embedding operation (depth increasing operand)

B. Operand of a concatenation operation (length increasing operand)

C. All other self-dependent occurrences

II. Non self-dependent

A. Constant (literal)

B. Variables without prior definitions (undefined uses)

C. All other non self-dependent occurrences

Case IIA: Constant.

Composite constants (such as constant sets or tuples) are built up from the individual constants using operators such as set or tuple formers. Therefore, the only constant occurrences are those whose types are scalars, i.e. **integer, real**, etc. The nesting depth of such an object is 0 and the length is -1.

Case IIB: Variable with no prior definition.

The type of this use will remain **error**, and it will be flagged as a use of an uninitialized variable. For example:

```
z := x + 5
```

Assuming there is no definition of x prior to its use in defining z, the typefinder results in a type of **error** for this occurrence of x. We can regard this as a constant (wrt to the typing process) whose implicit type is **error** and as such has a nesting depth of 0 and a length of -1.

Case IA: Self-

dependent and operand of an embedding operation (recursive, depth-increasing occurrence).

The typefinding algorithm assigns a new recursive type symbol to this occurrence whose nesting depth is (by definition) 0 and whose length is -1.

The above three cases have self-defined nesting depths and lengths. The (maximum) nesting depths and lengths of the other three cases can be computed by calculating their distance (in terms of statements) from an occurrence belonging to one of the previous three categories.

Case IB. Operand of a concatenation operation (length increasing operand)

The typefinder flags any cyclical length-increasing operands and transforms any sequence type symbol assigned to them into tuples (whose length is 0). With regard to nesting depth, they are the same as Case IC (below).

Case IC: All other self-dependent occurrences.

We examine each UD path leading backwards from this occurrence until we encounter an occurrence of type IA, IIA or IIB. If after traversing d nodes along such a path, we encounter such an occurrence, the nesting depth of the type symbol constructed along that path is at most d.

For example, given the following code sequence:

```
1  a := 5;
2  x := {a};
3  (while somecondition)
4    read (y);
5    x := x + {y};
6  end;
```

Examining the path:

$$\mathbf{x}_{5.2} \rightarrow \mathbf{x}_{2.1} \rightarrow \mathbf{a}_2 \rightarrow \mathbf{a}_1 \rightarrow \tilde{\ }5\tilde{\ }_{1}$$

we note that the (recursive) type structure of x has constant nesting depth of 1.

We must however, examine in some detail the possibility that we do not encounter one of the three kinds of self-defining occurrences but rather arrive back at our starting node (this is a definite possibility as we are dealing with a self-dependent occurrence). This occurs in our above example along the path:

$$\mathbf{x}_{5.2} \rightarrow \mathbf{x}_{5.1} \rightarrow \mathbf{x}_{5.2}$$

Note that if we are able to reach the occurrence in question by traversing a UD path, none of the occurrences along that path could be operands of an embedding operation, for were that the case, such an occurrence would be a self-dependent operand of an embedding operation, one of our self-defining cases, and our backwards traversal would stop at that point. Therefore, the path from the occurrence back to itself must contain no operations that increase the nesting depth of a type symbol (since we are able to traverse its length) and it follows that such a path can be viewed as having no effect upon type symbols (wrt their nesting depth). In the above example, for the cyclic path to increase the nesting depth of x, there would have to be some embedding operand along that path, but that would be a occurrence whose depth is self-defined. The same argument holds if during the traversal we encounter a self-dependent nonrecursive occurrence, distinct from the one being analyzed. The only paths that need be examined are the noncyclic ones. We see therefore, that full-cycle paths can be ignored.

Case IIC: All other non self-dependent operands.

The analysis is the same as Case IB, but we do not have to concern ourselves with a cycle as the occurrence is not self- dependent.

Note that in the above two cases, we assumed that eventually we would reach one of the self-defining cases. This must be the case, as there are only a finite number of occurrences satisfying Case IIC, and none of them can lead into a cycle (as they are not self-dependent). Furthermore, we are ignoring full cycles in Case IB and as such are only dealing with paths that are non-cyclic. **QED**

2.7. Completeness of the Algorithm

Our typefinding algorithm accepts the intermediate code and UD-chains of a SETL program and produces a set of type equations for the variables of that program. The method employed is quite similar to Tenenbaum's typefinding algorithm, the primary innovation being the typing of recursive structures which is made possible by assigning names to such composite types and distinguishing between the name and actual structure of such an object. The disjunction function now allows for the existence of recursive symbols. Given such a symbol, if the second operand of the disjunction function can be determined to be a subset of its internal structure, the result is merely the recursive symbol; otherwise the result is simply the alternation of the two inputs. For example:

dis(REC_1, integer) = REC_1 if REC_1 = integer | set(integer)

while

dis(REC_1, string) = REC_1 | string

Therefore, the disjunction function remains monotonic in nature and no alternands are lost when disjuncting two type symbols. Similarly, the propagation functions are the same as before, with the exception that recursive types are unfolded when necessary so that the propagation functions act upon nonrecursive symbols (i.e. integer, set(...), etc.). The only time a recursive symbol is not unfolded before being presented to a propagation function is when the operation is question is an embedding. In that case the symbol is merely enclosed in the appropriate constructor (e.g. set( ), or tuple( )) and no examination of the internal structure is necessary. What remains is to show that type information does indeed reach all points of the program.

Completeness, i.e. all type information reaches all occurrences, is guaranteed by the fact that each instance I keeps track of all other instances I' affected by any changes made to I.

Theorem: The structure occurrences_dependent_upon correctly propagates complete type information to any occurrences requiring that information.

Proof: There are three situations for which a variable occurrence must be placed into the workpile for processing due to a change in some type. These are:

1) The type of an ivariable is modified. In this instance, the ovariable of the instruction containing that ivariable must be (re)processed.

2) The type of an ovariable has been modified. The types of all subsequent uses (ivariables) of that definition must be recalculated to take into consideration the change.

3) As part of the propagation function of an operation, the internal structure of some recursive type symbol, R, must be examined. This occurs if R is an alternand of one of the ivariables of the operation in question. For example, in the statement:

(4) x := z + 3;

if one of the alternands of z_4 is R, when calculating the type of x_4. we must examine the structure of R to see whether one of its alternands is **integer**.

Situation 1 is handled by dumping the ovariable of an instruction into the workpile whenever the type of an ivariable of that instruction is modified. For case 2, the members of the DU set of an ovariable are placed into the workpile whenever the type of the ovariable changes. In case 3, we must show that if an alternand t, is added to the type structure of R, that information will be propagated to o prior to termination of the algorithm.

As shown above, the internal structure of a recursive type symbol R is examined when calculating the type of an ovariable in an instruction one of whose ivariables, say i, has R as an alternand. When R is initially made an alternand of i, o is immediately placed into the workpile (this is an instance of case 1), and when subsequently processed, o wil be inserted into the occurrences_dependent_upon set for R. Thus o will become part of the occurrences_dependent_upon set for R prior to algorithm termination.

We now show that an alternand, t, added to R's structure causes the type of o to be recalculated. If t is inserted into R's structure after o has been made placed into the occurrences_dependent_upon set for R, then the insertion of t (i.e. a modification to the structure of R) causes o (as well as all the other members of occurrences_dependent_upon for R) to be placed in the workpile for type calculation. On the other hand, if t is placed into R's structure prior to o's insertion into R's occurrences_dependent_upon set, then at some later point of the analysis (when R is made an alternand of i), o will placed into the workpile. When o's type is then calculated, t is already a part of R's structure and will therefore be taken into consideration in the calculation of the type of o. QED

Assuming that the propagation function act in accordance to our intuitive understanding of type propagation (i.e. that the semantics of the operation accept only certain types as legal inputs and based upon those types produce a result with some specific type), the result types generated by any operation will be correct. Furthermore, the output variables will be entered for processing whenever a type affecting them is modified, regardless of its location in the program.

(What we are essentially doing is constructing recurrence relations relating the types of the variable occurrences of the program. Unlike most other applications that deal with recurrence relations which must then solve the relations for a closed form, the recurrence relations that we construct are themselves the solution to the problem. For example, if we were to attempt to calculate the size of structures in a SETL program, we might introduce some length function, **len**. If the length of some structure within a loop was dependent upon its size in the previous iteration through the loop, e.g.:

```
(for i in [1 .. 10])
    read (x);
    t := t with x;
end for;
```

we set up the system:

$$\text{len}(x_0) := 0$$
$$\text{len}(x_{n+1}) := \text{len}(x_n)+1$$

where x_i is the value of x on the i'th iteration through the loop. We would then solve, in closed form, form x_{10} to obtain the length of x upon exit from the loop.

For the purposes of typefinding however, once the system of equations is constructed, they provide the solution to the typefinding, and it is not necessary to solve for any closed form.)

The auxiliary structure, **occurrences_dependent_upon**, that determines which occurrences are dependent upon which recursive symbols is a generalization of the UD and

DU chains traditionally employed in data flow analysis. Indeed, its size (if implemented as a bit string) is the same as these chains and in fact all three can be merged to form a new structure adding a touch more uniformity to the algorithm. Use-definition chains have long been the triggering mechanism for workpile algorithms in data flow analysis, and one can speculate whether such a generalization aids in the simplification or production of other data flow algorithms.

If the flow graph is traversed using R-postorder, UD chains are sufficient, as nonconvergence of a pass over the graph forces the visiting of every node another time.

2.8. Choice of the Recursive Instance

In the above algorithm, only the ivariables of embedding operations were made candidates for the self-dependency test and therefore for having recursive type symbols assigned as their types. The corresponding ovariables then receive a type dictated by the propagation function associated with the operation in question. We could alternatively test only the ovariables of embedding operations and allow their types to propagate back to the ivariables. The type equations that result are correct with respect to the program, but are quite different from those obtained by employing ivariables as the recursive instances. As an example in the construction of a linked list:

```
(1)    l := 5;
(2)    (while somecondition)
(3)       l := [5, l];
(4)    end while;
```

choosing the ivariables as recursive instances produces:

l_1 : **integer**
$l_{3.2}$: **REC_1**
$l_{3.1}$: **seq(integer, REC_1)**

where

$$\textbf{REC\_1} = \textbf{integer} \mid \textbf{sequence(integer, REC\_1)}$$

while having the ovariable ($l_{3.1}$) be the recursive instance results in:

l_1 : **integer**
$l_{3.2}$: **integer | REC_1**
$l_{3.1}$: **REC_1**

where

REC_1 = sequence(integer, integer | REC_1)

Both sets of equations produce the same domain of types for both $l_{3.1}$ and $l_{3.2}$, however, the equations themselves are quite different. The definition of REC_1 in the first case retains more of the flavor of a recursive definition of a linked list while the second case seems reminiscent of an undiscriminated variant record. Proving termination for the ovariable case is similar to the proof of termination presented above.

2.9. Typetesting Predicates and Typefinding

SETL provides a set of typetesting functions consisting of a function, **type**, that returns (as a string) the type of its argument, as well as predicates, **is_integer, is_real, is_set**, etc., to test whether an object is of a specific type. The information present in these predicates is not used in the present typefinder. This section explores the manner in which such information may be exploited.

The problem with using typetesting operations is that to employ information they provide requires knowledge not only of the basic block structure of the program (which until now we have been able to ignore, using UD chaining exclusively), but furthermore the true/false tagging of the successor block mapping (i.e. for a given successor of a basic block, is it a successor on a true branch, false branch or unconditionally?)

However, if this information is available we are able to get more precise type information in many cases. As an example, consider the following fragment that constructs a linked list:

```
(1)    t := om;
(2)    (while someondition)
(3)      if t = om then
(4)        t := [6];
(5)      else
(6)        t := [6, t];
(7)      end if;
(8)    end while;
```

l_6 is assigned the alternand **om** since the algorithm does not take into account the type information supplied by the conditional.

Propagating such information through the program is by no means a trivial task. The type test may have been performed several instructions back and its value passed through several intermediate objects. Furthermore, this information can only be exploited if the current boolean value used to perform a branch is definitely the result of a type test. If the test is conditionally performed the information is no longer valid. Similarly, if the results of two separate type tests merge, we may not be able to infer any information.

The above discussion indicates that if typetesting data is to be employed, it seems reasonable to limit the analysis to only those boolean values that are the output of a typetesting predicate and are immediately used for a conditional branch (e.g. **if is_om t then ...**). Examination of SETL programs indicates that in any event these are the most frequently occurring uses of such predicates.

2.10. The Backwards Pass

The above algorithm operates in a forward fashion assigning types to the occurrences of a program and as such replaces Tenenbaum's forward pass. We now examine the backwards pass whose goal is not to discover strong typing but rather to insert type consistency checks.

Given two recursive type symbols, REC_1 and REC_2, one can always define their disjunction (which is the operator used in the forward pass) as REC_1 | REC_2. The possibility that the two types may be identical (or overlap) in their internal structure does not make the union incorrect, merely redundant. One can apply a structural equivalency algorithm, such as the one used for testing type equivalence in Algol68 [Kost69], to determine whether the two are one and the same, but this step can be postponed to a postprocessing phase after the forward pass has completed.

The backwards pass however uses the meet (or intersection) operator. Taking the intersection of two recursive type symbols requires the examination of the structures of both symbols to determine what alternations they have in common (if any) and this operation cannot be postponed. That is, when it is necessary to take the union of two recursive types, REC1 and REC2, to calculate some type t, if the internal type structures of REC1 and REC2 are ignored and we assign to t the type REC1 | REC2 (i.e. we do not perform a structural equivalence test), the worst that happens is that t contains duplicate alternands (e.g. **tuple(integer)** | **tuple(integer)**). On the other hand, when it is necessary to perform a conjunction (intersection) of two recursive types in order to calculate some type t, we must examine the internal structures of the two types in order to determine what alternands they have in common, as those are the only ones to be assigned to t. To overcome this difficulty of comparing recursive types for purposes of conjunction, there are several alternatives to the manner in which the backwards pass might be handled.

First, we can keep the backwards pass unchanged. As the current version has no notion of recursive type symbols, all that is required is that as a preprocessing phase all such type symbols be set to **general**. No information is lost over Tenenbaum's forward pass as any such occurrences are assigned **general** by that algorithm.

Alternatively, we can attempt to perform intersections involving recursive type symbols. Conjuncting a recursive with a nonrecursive symbol requires checking whether the nonrecursive symbol is an alternand of the recursive one. Intersecting a recursive symbol with itself is trivial. In order to intersect two different recursive symbols we can either perform the structural equivalence test mentioned above, or, more simply, revert to **general** (which is what would occur in the Tenenbaum version).

The above discussion assumes that the detection and construction of the recursive data types had already been accomplished by the forward pass. A separate issue affecting the backwards pass is examination of programs in which the forward pass produces no recursive structures. Given large-scale systems involving several phases, intermediate data structures may be written out to mass storage devices by one phase and later read in by a subsequent phase. In such cases, the structure is used (and not defined) by the second phase and therefore the forward pass, which determines types from definitions, produces no information about the structure. As an example, a compiler whose syntactic and semantic checks are performed in separate passes might build the parse tree in the syntactic phase and write it out to a file. The semantic phase, which is to annotate and validate the tree would then have the initial statement:

```
read (parse_tree);
```

The forward pass has no choice but to assign **general** to parse_tree.

The question remains whether the techniques introduced in the forward pass, namely the separation of name and structure, can be incorporated into the backwards pass as well and thus allow for the typing of recursive structures by examination of the manner in which they are subsequently used. As the processing is done in a backwards fashion, extracting components rather than constructing structures, it may be necessary to introduce notation and symbols into the type lattice to allow one to express the fact that t is the type of the i'th component of the sequence t'. We might then apply a method similar to that used in the forward pass, i.e. finding all self-dependent instances, and rather than propagating actual component types through the program, keep that information in a separate structure and instead propagate the relationship of a component to the tuple from which it was extracted. Thus, if a self-dependent instance, i, is the third component of some tuple t, use as the type of i the symbol t(3) (i.e. the type of i is the type of the third component of type symbol t), and collect the actual type of i (e.g. integer) in a separate structure. We plan to examine this area in the immediate future.

2.11. Comparison with Other Algorithms

2.11.1. Tenenbaum's Typefinder

As seen above, the algorithm presented in this thesis performs substantially better than Tenenbaum's in the presence of recursive data types, and produces more precise type information. The actual processing of variable instances is performed in an identical fashion to Tenenbaum's algorithm (with the exception of examining internal structures of recursive type symbols which we discuss below) and this method has been shown to require an arbitrary number of iterations over the instances of the program [Shar78]. In addition, with regard to the number of iterations over the program, our algorithm does no worse than Tenebaum's, and indeed depending upon the arbitrary nesting limit may converge in fewer iterations. This is because the introduction of recursive symbols forces immediate convergence of the type of the ovar of the statement in which such a symbol occurs, whereas Tenenbaum's algorithm requires the appropriate depth to be reached before convergence (through transformation to **general**) occurs. For example, in the following fragment:

```
(1)    l := 5;
(2)    (while somecondition)
(3)       l := {l};
(4)    end while;
```

our typefinder results in the following processing sequence:

l_1 : **integer**
$l_{3.2}$: **REC_1**   **$ REC_1 : integer**
$l_{3.1}$: **set(REC_1)**
$l_{3.2}$: **REC_1**   **$ REC_1 : integer | set(REC_1)**

following which the algorithm converges. On the other hand, given a limit of 3 upon nesting depth, Tenenbaum's algorithm produces:

l_1 : **integer**

$l_{3.2}$: **integer**
$l_{3.1}$: **set(integer)**

$l_{3.2}$: **integer | set(integer)**
$l_{3.1}$: **set(integer) | set(set(integer))**

$l_{3.2}$: **integer | set(integer) | set(set(integer)))**
$l_{3.1}$: **set(integer) | set(set(integer)) |**
    set(set(set(integer)))

$l_{3.2}$: **integer | set(integer) | set(set(integer)) |**
    set(set(set(integer)))
$l_{3.1}$: **set(integer) | set(set(integer)) |**
    set(set(set(general)))

$l_{3.2}$: **integer | set(integer) | set(set(integer)) |**
    set(set(set(general)))

at which point the algorithm converges. Had the nesting limit been larger, the nonconvergent variables $l_{3.1}$ and $l_{3.2}$ would have been processed a larger number of times.

Of course, there is the additional overhead of testing for self-dependency in the initialization phase and collecting the internal structures of recursive type symbols when it is necessary to examine them. The first of these is quadratic with respect to the number of variable instances in the program (i.e. every operation may be a self-embedding one requiring the self-dependency test to be performed upon all ivariables in the program). The second may in the worst case (again all ivariables are self-dependent) require examination the structure of all recursive type symbols (which can be as many as the number of ivariables in the program) in the program each time an ovariable is processed. Nevertheless, in actual programs the number of recursive type symbols generated is quite small (example 9 of Chapter 5 which is a simple parser requires 4 recursive symbols), and therefore the overhead seems reasonable.

If the check for self-dependency presented in [Fong75] is employed, information regarding recursive structures is even worse than that derivable from Tenenbaum's original algorithm. This is because the discovery of a self-dependent embedding operand causes it to be assigned **general** prior to actual typefinding and thus no information regarding its structure can be obtained. Without this enhancement, the algorithm does produce some additional (though incomplete) information regarding the type (see the above example).

2.11.2. Type Unification

In contrast with the weak typing of SETL, B and ML are strongly typed; this is the primary reason for the different approach to typefinding which is used in these languages. As mentioned earlier, type unification consists of intersecting possible type templates of variables to insure that they are consistent, while the initial pass of SETL's typefinder uses union as its basic operation. This is due precisely to the differences in the type models. We can think of type unification as being equated with the second (backward) pass of the SETL typefinder, the chief distinction being that whereas B and ML use the derived information to flag type errors, SETL can do no more than use the backward pass to insert consistency checks due to its weak typing. Furthermore, while SETL makes a distinction between the forward and backwards passes, the forward typing via definitions, the backwards via usage, in B and ML a definitions and usages are used simultanaeously to produce type equations that are to be unified.

To summarize in a different light, B and ML, have no need to deal with the union of disparate types at the entrance to a block and do not require a disjunction operation. Furthermore, since they need only check for strong typing, they need not uncover possible types (through a forward pass that propagates definitions) but rather ensure type consistency. It is therefore clear that the backward pass of the SETL typefinder plays the same role as the unification algorithm with respect to what it uncovers. The forward pass is there because of the weak type model. On the other hand, it is precisely the weak model that allows SETL to support the recursive structures we have been examining. B is unable to do so because the variety of shapes that such structures assume violates its strong type model. ML does allow recursive structures but only in the presence of explicit definitions.

In terms of complexity, it therefore clear that SETL's typefinder is by necessity more expensive.

2.12. Applicable Optimizations

Typefinding of recursive data structures allows us to determine the data types contained within the data structure and the internal structure and relationship among the individual data elements comprising the structure. This allows an increased number of variables in the program to be strictly typed.

Tenenbaum's typefinder, when presented with a program containing recursive data structures, types those structures as **general**. In addition, all variables directly, or indirectly deriving their values from such a structure are also made **general**. Thus, in a application working with a tree, all nodes including the leaves become **general**, as well as any objects manipulating the data elements at the leaf level. Note that the leaves themselves may very well be strongly typed, but as the internal nodes of the tree (represented by arbitrarily nested tuples) must go to **general**, and the leaf type is itself an alternation of the tree type as a whole, the leaf will be absorbed into the type **general**.

Once we are able to type the tree, the escape clause (i.e. those alternands of the recursive type symbol which do not contain recursive type symbols), which corresponds to the leaf type, gives us strong type information concerning the leaves and, indirectly, all variables extracting values from the leaves. In our binary tree typefinding example, objects deriving their value from leaf nodes can be typed as **integer**.

To examine a simpler example, consider the following fragment:

```
l := 3;
(while somecondition)
  l := [3, l];
end;
```

Tenenbaum's algorithm results in:

$l_{3.2}$: **integer | tuple(integer, general)**

which, though it does allow typing of the data element of the list (i.e. the first component of the tuple), provides this information only for the top level node. All information concerning the second component is lost and furthermore, variables assigned the second component (i.e. a variable later used to traverse the list) are typed as **general**.

Our algorithm however provides much more precise information:

$l_{3.2}$: **REC_1**

where

REC_1 = integer | sequence(integer, REC_1)

and therefore no information concerning the type or structure of the linked list is lost. The next subsection discusses to what advantage the optimizer can use this information with respect to generating more efficient code.

The difference is even more pronounced when the program contains trees and more complex structures. In the instance of a tree, for example, the return value of the construction subroutine (see example 7 of Chapter 5) is either a leaf, a node with a right son, a node with a left son, or a more general subtree. In the last three cases the representative tuples are of differing lengths, the general subtree and right son cases being of length 3, and the right son case of length 2. Tenenbaum's disjunction function unifies such fixed length tuples into a single unknown length tuple, whose component type is the disjunction of all components of the input tuples.

The effect of performing typefinding upon recursive structures can be summed in a simple statement: type **general** is no longer introduced into the type equations of a program except through the **read** statement or the use of external routines that have no type specification for their parameters. Previously, type **general** appeared in a program whenever the recursive structures (or constructs resembling such structures) were present.

2.12.1. Storage Applications

The current SETL implementation is interpreter-based with little emphasis given to storage efficiency. In a more static environment, the precise typing provided by recursive typefinding can result in more efficient representations.

Currently, if a variable can be strictly typed, in-line code can be generated for operations upon that variable. Otherwise, a run-time descriptor capable of denoting any SETL object is carried along with the variable and is used to determine which operation is appropriate to the object the variable currently contains. If it can be determined that a variable can contain at most two types of objects, rather than maintaining the full run-time descriptor, a single bit can be maintained and the test for which operation to be performed can again be generated in-line.

When a tuple is allocated at run-time, extra storage is assigned to it to allow the tuple to expand. Recursive typefinding provides us with the capability to determine that the tuple representing a recursive structure (e.g. a tree) is of fixed size and therefore, the exact amount of storage can be allocated to it.

Finally, the fact that we are able to determine that a structure is uniform (e.g. in the case of a tree, each level consists of a tuple of three components, the first component an integer, followed by two tuples of the identical structure) may allow for more efficient code generation techniques and run-time storage management. We hope to examine this area of research in the future.

2.13. Addendum: The Advantages of Not Assigning Recursive Symbols to Ovariables of Embedding Operations

The initialization phase of the algorithm only assigns a recursive type symbol to ivariables that are members of their own BREACHES* set. This is because flagging ovars as recursive will produce additional types that are unnecessary, as they are nothing more than subclasses of their right hand counterparts. For example, in the following code fragment (taken from section 2.6):

```
1    t := 3;
2    (while somecondition)
3         t := [t];          $ t embeds itself
4    end while;
```

the ovariable $t_{3.1}$ above is nothing more than REC_1 without the scalar type **integer** alternand. Since we are concerned with recursive structures that are constructed with embedding operations, it is clear that a recursive ovariable whose defining operation is an embedding operation will not be able to assume a primitive leaf type. Consider the above example; if the ovar $t_{3.1}$ was assigned its own type name, its structure would be **sequence(REC_1)**. First, this gains us nothing, as that is precisely what the above analysis provided us with but employing one less recursive type (we will see that this is desirable); second, this type is nothing more than REC_1 without the integer leaf type. Furthermore, these are two instances of the same variable and they require subsequent merging, a task made more difficult by the introduction of the second type name.

To see this more clearly, if we type both self-dependent ivariables as well as ovariables in the above example, we produce:

$$\begin{aligned} \mathbf{t_1} &= \mathbf{int} \\ \mathbf{t_{3.2}} &= \mathbf{REC\_1} \\ \mathbf{t_{3.1}} &= \mathbf{REC\_2} \end{aligned}$$

where

$$\begin{aligned} \mathbf{REC\_1} &= \mathbf{int} \mid \mathbf{sequence(REC\_2)} \\ \mathbf{REC\_2} &= \mathbf{sequence(REC\_1)} \end{aligned}$$

which while correct, is murky.

We see therefore that the self-dependency test need only be applied to ivars, as those are the only occurrences to which we assign new type symbols.

We next note that the only occurrences that truly need to be assigned new type names are those that are actual operands to the embedding operations. Occurrences that are simply assigned the embedded value along the circular path from use back to itself do not introduce any new levels of nesting and thus pose no problems. It is only those operands of these embedding operations that produce result types that may not converge. Eliminating the non-convergence in these instructions will therefore remove the problem in general.

The above reasoning can be made clearer if we borrow from the terminology of hardware arithmetic. Ivariables whose associated operations are nonembedding can be thought of as nonconvergent propagators, that is, they cannot introduce a new nonconvergent instance into the program, but will propagate the nonconvergent instance's existence if they are assigned its value. Operands of embedding operations, on the other hand are nonconvergent generators, capable of producing new nonconvergent types. If we prevent the generators from producing such types, the propagators present no problem. Therefore, it is only the operands of embedding operations that have to be considered for new type names.

CHAPTER 3

Explicit Typing in SETL: The Representation Sublanguage

3. Explicit Typing in SETL: The Representation Sublanguage

Although SETL is declaration-free, a **data representation sublanguage (DRSL)** is provided to allow the programmer to inform the system as to the types of variables. Using this information, the compiler can generate appropriate in-line code for some of the operations whose operand types are known. The representation sublanguage also provides the user with the facility to choose particular internal representations for some of the language-defined composite types. Thus, even in the absence of any typefinding optimizations, the programmer can achieve a speedup in his code.

These two facilities provided by the DRSL are logically independent of each other. Thus, it is quite reasonable to have a programmer specify type information and omit data structure representation; and conversely structural relationships (e.g. which variables are used to access the maps in a program) among variables in a program can be provided without giving any clue to the actual types assumable by those variables.

The typing and structuring facilities also require different levels of knowledge, both of general data structures and internal SETL representations, on the part of the programmer. The typing representations perform the same function as declarations in lower level languages: binding of variable and type at compilation for code generation purposes and compiler type checking for program reliability. The data structure representations, or **basing declarations**, allow the programmer to specify, *at some time after the algorithm has been written and tested*, the internal representations composite objects and their elements should assume.

(We coin the term **repering**, to denote the act of declaring an identifier to be of a specific type. The token **repr** is used to signal the beginning of the optional declarative section of a SETL program.)

3.1. Basings

In addition to type specification, the data representation sublanguage provides the programmer with the facility to declare relationships among objects in a SETL program. This capability can be divided into two areas: elaboration of access relationships between the various objects; and specification of the internal representations of composite objects (i.e. sets, tuples and maps).

The standard data structure used for maintaining sets in the SETL run-time system is the hash table. Elements are hashed using some system-wide hashing function, and placed in the corresponding bucket of the table. Membership testing as well as most other set operations therefore require a hash as well as a possible traversal along the bucket's chain.

To eliminate the ubiquitous hashing, the representation sublanguage provides the programmer with the ability to set up universes of objects and subsequently declare program entities as being elements or subsets of these universes. For example, given a program manipulating graphs, a universe of nodes may be declared (possibly of a specific type, e.g. **atom**), with successor and predecessor relations **based** upon this universe. Furthermore, noncomposite entities (e.g. variables used to hold single nodes) may be declared to be elements of the universe of nodes. The above is expressed as:

```
base NODES;
successor : map (elmt NODES) elmt NODES;
x         : elmt NODES;
```

The internal representation of the universe (or base) is also a hash table containing all elements of the base. When an entity, say x, declared as an element of the base is assigned a new value, that value is inserted into the base (if necessary) and a pointer to the value's entry (or **element block**) in the universe table is assigned to x (rather than the value itself). To access any information in the table entry for x (e.g. the value of some mapping f(x)), only a dereferencing operation need be done, as opposed to a hashing operation.

In order to take full advantage of the basing facility, the programmer is able to further specify the internal form of any sets or maps that that have the base as their domain or range. There are three such representations: **local**, **remote**, and **sparse**.

1 The **local** format stores the characteristic bit (the characteristic bit of an element in a universe wrt some set is 1 if the element is a member of the set and 0 otherwise) for a set directly in the table entry of an element in the base. a) To test an object x, which has base b, for membership in set s, which is declared as based upon b, the pointer in x needs only be dereferenced and the appropriate bit in the element block of x be checked. b) For maps, the range value is maintained in the element block of the corresponding element of the domain.

2 A field is statically reserved in each element block for an object declared **local**, and so entities made a part of other objects are precluded from being represented in such a fashion since no other object can share that field. Furthermore, set operations, such as union and intersection are cumbersome to perform when sets are represented in local format. To alleviate this, an alternative representation, **remote**, is supplied which provides an array-like representation. Each object inserted into the universe is assigned a unique number in sequential order. When an entity is declared remote, an array (whose size is that of the base) is allocated. The entry for the object whose assigned number is i is maintained at cell i of this array. This entry is a bit for sets and the range value for maps. Using this representation, set operations are trivially carried out via bit string operations.

3 Finally, there is the default representation, **sparse**. This format is identical to the unbased representation (i.e. a separate hash table for the set or map), but the entries in

the hash table are pointers to the appropriate element blocks in the base.

3.2. A Critique of the Representation Sublanguage

There are two levels at which a critique of the representation sublanguage can be directed. First, there is the conflicting goal of utilizing the power of SETL's weak type model in achieving a natural expression to the solution of a problem vs. the strong typing required for repering and efficient code generation. Second, there are specific problems posed by the current representation sublanguage: incompleteness, lack of local bases, etc. (many of which are direct consequences of the first and more general problem). We first address the former and then discuss the more specific problems together with some proposed solutions. Emphasis is on the typing aspect of the representation sublanguage as opposed to the data structure selection facility.

3.2.1. Reprs and Strong Typing

At the forefront of the repering problem is the fact that SETL is a weakly typed language supplemented by a representation sublanguage which, as it currently stands, attempts to impose strong typing. The basic motivation for performing automatic type-finding in a weakly typed programming language is that *good* (i.e. structured) programming style does not use the same variable for different purposes and surely not for objects of different types, and therefore the majority of variables can be assigned a definite and unique type.

This argument, while true in a majority of situations, most notably those for which the SETL coding is very close to the corresponding program in some lower level, more conventional language, does not apply to algorithms which involve entities that are polymorphic. It is precisely in such instances that the power of SETL is exploited in expressing the algorithm in its natural fashion. A clear example of this appears in the prototype Ada/Ed translator [Kruc84]. Given a name (identifier) at a specific point of an Ada program, the corresponding entity might not be unique (as a result of overloading), and thus can be either a single entity or a set of such items. The obvious approach in SETL is thus to allow the variable designating that entity to be either the unique name of the entity in question, or a set of such unique names. Such a variable must then be repered under the current system as **general**. Indeed, to address the above problem in such a way as to permit repring would defeat much of the purpose of using SETL in the first place.

Another area in which weak typing is natural is that of generic procedures. There is no reason why a SETL program should have more than one sort procedure, regardless of the components types of the tuples that have to be sorted (the overloading of the comparison operator on numeric and string types makes this possible). Sorting a tuple of numbers or a tuple of strings is syntactically identical, except for the declaration of the component type of the tuple (a fact which often motivates the introduction of generic procedures into otherwise strongly typed languages, Ada for example). Since 'pure' SETL does not include declarations, having two such routines would be totally redundant. Of course the design of a system might take into account the eventual introduction of reprs into the system, and thus incorporate two routines, but this would be in violation of the concept of using SETL as a means of ignoring implementation questions.

One of the fundamental goals of the data representation sublanguage is that declarations must be semantically neutral from the rest of the program. That is, the design of an algorithm in SETL should avoid having to take into account typing and data structure issues. If such issues are incorporated into the design of the program, much of the appeal of SETL as a prototyping language is lost. A conscientious user is far less inclined to take advantage of the weak typing or absence of declarations in SETL if using such features is done at the cost of program reliability. Optimally, these issues should be later addressed via the DRSL. However, from the above discussion it is clear that in order to be able to strongly type variables under the current data representation sublanguage, some degree of forethought (with respect to typing) has to go into the design of the algorithm. To do so however is to steer oneself away from thinking in terms of SETL and designing programs at a level closer to PL/I or Algol. Generic style routines, variable length as well as varying format heterogeneous tuples and arbitrarily nested tuples might be avoided, all because there is no facility to specify such data types in the current system.

Part of the above problem is due to SETL having a natural syntax. Thus, operator overloading is extensive (e.g. + is used for addition of integers and reals set union and string and tuple concatenation). Maps and tuples are manipulated in a syntactically identical fashion, requiring a reader to trace back to the structure's initial assignment to see its type. Though these facilities are a great convenience to the programmer, allowing him to avoid exact type decisions to a large degree, they have a detrimental effect upon static type validation.

The above issues can be succinctly expressed in terms of the underlying type model of SETL. As it currently exists, the data representation sublanguage is capable of expressing only a proper subset of the full type model of SETL. With the exception of type unions and recursive structures, the majority of those structures that cannot be directly typed do not frequently occur. (For example, the case of a tuple consisting of a single string followed by an arbitrary number of integers cannot be strictly typed, even in the presence of an alternation operator. However, most programmers would not use such a structure anyway, but rather a two component tuple, the first element being a string, the second a tuple of integers.) In addition, many of these structures are too complex to gain any efficiency from strictly typing them, as a run-time check on their structure is required regardless. However, typechecking has in recent years been recognized as having a second and equally important function: enforcing a reasonable type structure upon the programmer in order to increase program reliability. In light of this, the attitude that any type mode which cannot be specified in the DRSL, but which exists in the actual type model must be typed as **general** may be reasonable for code efficiency depending upon the actual implementation (since any object which has an alternated type must in any event be checked for its current type at run-time), but for purposes of rigorous type validation precise typing facilities is desirable. For SETL and the DRSL to be logically independent in the fullest sense, at the same time providing the user with a well enforced type environment, the DRSL must be capable of expressing all reasonable type modes that can arise at run time. Again, this capability should be provided even in situations where the only beneficial result is in terms of program documentation (i.e., no code generation optimization can be performed and thus the DRSL types the object as **general**).

This inability to specify certain type structures is felt strongest in the areas of type unions and specification of recursive structures. Names are occasionally polymorphic due to the nature of the algorithm and there should be a facility to specifiy the valid sets of values assumable by these names. Given the ability to specify a type using a recursive declaration, recursive data structures can be given clean and uniform declarations, as opposed to having to declare them as **tuple(general)**.

There is, in addition to the above, an underlying problem to the question of type specification using the representation sublanguage, and that is the degree of skill required to use it properly. Directing our attention to the internal representation facility, we see that there are no clear guidelines as to what representation to assign to a particular data structure. (In fact, in the automatic data structuring facility, the criteria upon which to base the choice as to which of the three representations, **local, remote** or **sparse** to choose are coarse, frequency of access and object size information currently being unavailable). However, one expects that type specification should be a straightforward task, which if would not speed-up the code, would at least not slow it down. Yet such does not seem to necessarily be the case. Consider the following program structure:

```
program test;

read (a,b);
s(a,b);
```

```
procedure s(a, b);

if x = 0 then
   t1(b);
else
   t2(b);
end if;

end procedure s;
....
{procedures t1 & t2 which each contain
     uses of formal parameters bound to b}
....

end program test;
```

Assuming that the values of a and b are to be integers throughout, one should be able to include:

```
repr
   a  : integer;
   b  : integer;
```

in the main program and each of the routines and expect a speed-up in each case. Note that routine s merely acts as a pipeline for variable b and involves no uses of its value which would require a run time check. Furthermore, programmer type specification of a variable causes type checking to be moved from point of usage back to the point of definition. Therefore, declaring a and b to be integers in the main routine causes an insertion of a type check at the read (for the main program) and at the initial parameter assignment (for s) whereas prior to repering no typechecking needed to be performed for b in either of those two routines as they involved no use of b that required such a check.

It would thus seem that under certain circumstances, type declarations could yield needless code. To avoid this should not be the responsibility of the user but rather that of the optimizer itself (e.g. check for uses of that variable and if none exist, delete the type check at definition). As mentioned previously, type declarations have a documentation purpose that is necessary to the reader, and thus declarations should not be omitted simply for the sake of object code efficiency.

3.3. Extending the Representation Sublanguage

As seen above, the current facilities provided by the representation sublanguage are inadequate for the expression of many of the data structures that SETL is particularly suited for. For example recursive relationships among variables cannot be expressed. An extension to the sublanguage must therefore allow for such specifications.

In the following discussion, we distinguish between changes in the DRSL that require a change in the SETL programming language proper, and changes that can be effected entirely within the data representation sublanguage. In all instances however, we require that specifications made within the DRSL be transparent with respect to the original SETL code.

3.3.1. Recursive Specifications

The first part of this thesis presented an algorithm that automatically types and discovers the internal structure of recursive data types. It is therefore only natural that the programmer have the capability to declare such objects using the representation sublanguage. We introduce a new operator into the DRSL, alternation, |, which allows one to declare an object as capable of denoting more than one data type. In the current DRSL, such an object has to be declared as **general**. The following section deals with the generalities of alternation; for the present we are merely interested in it as a means of specifying a recursive type.

Recursive types must be declared as modes, i.e. they may not be directly defined as the type of a specific variable. To declare a binary tree, for example, one writes:

```
mode tree : integer |                          $ leaf type
            sequence(integer, tree, tree)      $ internal node
```

(**sequence** denotes a fixed length tuple). This mode can then be used in the same manner as any other, for variable or parameter declaration:

```
parse_tree    : tree;
semantic_pass : proc(tree);
```

Mutually recursive definitions are also allowed. For example:

```
REC1 : integer | sequence(integer, REC2)
REC2 : sequence(string, REC1)
```

(See example 2 of Chapter 5 for a program in which these types appear.) The only restriction is that there must be one definition in the chain that contains an alternation without a recursive reference.

3.3.2. Type Alternation

Alternation has other uses besides providing the DRSL with the capability to specify recursive data types. With respect to the general problem of weakly typed variables, it seems that some means of specifying a list of alternative valid types is in order. The chief advantage of typing a variable is in the fact that one can then verify the validity of assignments to that variable and thus subsequent uses of the variable can be assumed correct; in addition, operations involving it may be determinable at compile time allowing for the generation of actual machine code to perform the operation Allowing a list of valid types might still reduce some of that run-time overhead, (depending upon the implementation of the run-time routines) possibly at the expense of having several additional in-line tests. At the same time it is appropriate to have a maximum number of allowable alternatives supported and have the system treat any variables with more than that number of possible data types as being of type **general**. Two alternands seem to be t maximum number that would gain by placing the code in-line. Finally, in terms of program reliability, giving the programmer the capability to specify the exact set of alternative types a variable can assume allows the compiler to perform some degree of type checking. For example given the following SETL fragment:

```
x := func1;
    .
    .
y := func2;
    .
```

```
        .
    z := x + y;
```

If func1 and func2 are external routines (and therefore without type specification from the programmer must be typed as **general**) and furthermore func1 returns either an integer or a set of integer, while func2 returns either a character string, the system can flag the assignment to z as a definite error, assuming that type alternatives can be expressed. The current declarative system, however, requires x and y to be specified as being of type **general**, and thus any information about the type(s) of x is essentially lost.

Furthermore, in following the value and type flow of variables throughout a large scale system, it is useful to maintain specific information (rather than immediately resorting to **general**) as it may later allow one to collapse a list of alternatives back into a single type along a particular path in the program. The Ada/Ed compiler is a good example of a program where such a facility, even if only for documentation purposes would have been useful. As mentioned above, due to the overloading facility in Ada, names and operators are ambiguous until resolution. Thus variables that denote the unique name(s) of an identifier are either simple strings denoting the unique name corresponding to that identifier, or sets of such strings. Thus the function whose task is to determine the unique name(s) of a particular instance has to be repred as returning a type **general**. If upon returning from this function, the invoking procedure performs a set extraction upon the returned value, having the ability to specify the return type of the function as being **string** or **set(string)** informs the reader immediately that the extracted element is a string, whereas a return type of **general** tells us nothing.

This notion of type alternation is similar to the type union of Algol 68. In both instances, the types are **free unions**, i.e. there is no explicit tag field to control the current type of the object, but rather the system maintains its own internal tag. In Algol 68, however, the programmer is required to place any code manipulating such a type in a case statement with each possible type assumable by the object handled by one of the cases. This cannot be done in SETL without violating the transparency of the DRSL, as a programmer need not declare the type of an object at all, and thus the system has no way of knowing whether all possible cases have been covered.

3.3.3. Polymorphic Procedures

Many algorithms accept a data structure as a parameter and operate upon it in identical fashion regardless of the type of the data elements contained within the structure. Such algorithms include sorting and searching methods, various data structure primitives such as push and pop, etc. In each instance, the problem is that of manipulating what may best be called the 'location' of the datum rather than its actual value. It is clear that the binding of such an algorithm to the type of the informational element is not dictated by the guidelines of strong typing. Several languages, Ada among them, have acknowledged this fact and as such, have incorporated into the language design a facility to express such type independent algorithms in a manner that does not clash with the rest of the strong typing model.

These **generic** procedures, as they are commonly known, are a convenience to the programmer as they eliminate the need to maintain multiple copies of routines that are identical up to of the data type they manipulate. In addition, in terms of system documentation and structure, a single source routine is cleaner to an overall perception of the system. However, when the question of implementing such a routine arises, the design goals of a language that imposes strong typing must be taken into consideration. Besides helping to maintain the integrity of program variables, strong typing allows for the generation of efficient object code, in which the run-time representations of objects together with the code sequences corresponding to operations can be determined at compilation. Generic routines are therefore not implemented as type-free blocks of code whose data elements and operations remain typeless until run-time. Instead, the generic capability can be viewed as an enhanced macro facility relieving the programmer of the redundant work of recoding nearly identical routines.

3.3.3.1. The Generic Facilities of PL/I and Ada

PL/I offers what it terms a generic capability through its **GENERIC** attribute, although in actuality the facility is extremely limited. An identifier is designated as a **GENERIC** entry and acts as an alias for several subroutines whose parameter lists differ in type and/or number of arguments. It still remains the programmer's responsibility to write the individual routines, while the translator determines from the calling argument list which is the appropriate routine to be invoked. Thus the macro replacement occurs only at the point of call. System maintainability is therefore for the most part as it would be without the **GENERIC** declaration. The primary advantage is to allow for a uniform invoking name for several logically related algorithms (e.g a single calling identifier for several I/O routines whose parameter types are different).

Ada provides a more comprehensive generic facility, one in which types and operations can be manipulated as limited first class entities even though the remainder of the language disallows such manipulation. The programmer writes the algorithm, leaving as symbolic parameters those types and operations which vary between invocations of the procedure. When the algorithm is required for a particular data type, it is instantiated with the type and any associated operations that are required for actual compilation and subsequent execution. At the source code level, there is only one copy of the algorithm, while generated object code might contain one copy per instantiation. Type checking is only needed once, at definition. All usage is guaranteed correct.

3.3.3.2. Polymorphic Procedures in SETL

For strongly typed languages such as Ada, the generic facility acts as a sort of widening operator, allowing the programmer to express type independent algorithms without violating the type model of the language. SETL obviously requires no such facility as the types of parameters are not bound until run time and therefore any algorithm is in essence generic as long as the operators used in the routine are consistent with the type of the actual parameters. A generic routine, however can be viewed from the standpoint of SETL as a narrowing operator, allowing a controlled and limited use of type **general**. Currently, any routine which could be given diverse parameter types requires a declaration of **general** for its formal parameters. However, there is a loss of information involved in doing this. Consider a routine which is to sort tuples of integers or reals. Declaring the input tuple as being of type **general** necessitates full run time type checking. In particular, when comparing two elements of the tuple, a check must be performed that an ordering exists for their types. If the input type to the sort procedure is instead declared to be either a tuple exclusively of **real** or one of **integer**, the run time test for type compatibility of tuple elements can be eliminated. With the addition of an alternation operator, the semantic ability to express this would exist in a restricted fashion in the data representation sublanguage. As an example, a sort routine whose input is either real or integer could be declared as:

sort : procedure(tuple(real) | tuple(integer))

The semantics of this form, however, does not allow for a full generic capability. If the sort routine can be used for any type of input, there will be no way of expressing that fact. Furthermore, there will be no way to bind the types of local variables (such as a temporary) within the routine, whose values are derived from the formal generic parameter to that of the actual parameter. As an example of this problem, if a generic bubble sort is written, it is desirable to have the temporary variable used in the swap declared as being of the same type as the component type of the array.

A solution is to introduce a new type, which we call **generic**. This type is not a new element of the underlying type lattice, in fact in terms of the lattice it is identical to type **general**. Logically however, **generic** has an additional constraint: the alternation is exclusive, that is, a compound object can have as its component type any valid type, but once a component type is selected, all other types are excluded for the rest of the lifetime of the object, i.e. once the object is destroyed, upon exiting a procedure, for example, the object, when next created, can again be assigned any type. This notion corresponds somewhat to the typing rules of the B language in which objects are strongly typed by assignment. Generic can be used directly in a variable declaration or to define a new mode. The former use is sensible only for sets, maps and tuples and denotes the fact that component types can be anything, but must be homogeneous. The latter usage provides the capability of relating the types of several variables without actually specifying the types (somewhat similar to polymorphic type variables). Thus, the above sort example can be rewritten:

```
mode arr_element : generic;
sort : proc (tuple(arr_element));
```

and as a local declaration in the sort procedure:

```
temp: arr_element;
```

When used in the mode form, assigning an object to a variable declared to be of that mode restricts the type of the mode to the type of that object for the the lifetime of the procedure in which the mode is declared. Thus, if within the procedure sort we have the declaration:

```
mode arr_element : generic
```

the mode arr_element exists for the lifetime of the procedure sort. The first assignment of an object of type t, to a variable declared as arr_element restricts the type of all other assignments to variables of mode arr_element to t. When sort is exited, the mode is destroyed, and if sort is subsequently reentered, the type of arr_element is again open to an assignment of any type object.

In order to allow the parameters of a procedure to be related via a generic mode, mode declarations can now precede the parameter declarations in a procedure specification. Thus, to specify a procedure, insert, which accepts a tuple of some arbitrary type, and an element of that same type we write:

```
insert : proc (mode list_element : generic,
                    tuple(list_element),
                    list_element);
```

The mode list_element is visible within insert and exists for the lifetime of the procedure.

In the sort example, a translator can then use the knowledge that all array elements as well as the temporary are of the same type to eliminate the type compatibility test between components of the input tuple.

3.3.4. Exclusive Alternation

As noted previously, the programmer should have the ability to distinguish between a tuple with components of **integer** or **real**, and either a tuple of **integer** or a tuple of **real**. Though the representation sublanguage allows for such distinctions, the transformation functions of Tenenbaum's typefinder combine the latter two tuples into the former. As seen above, the programmer could declare the two cases as:

tuple(integer | real)

and:

tuple(integer) | tuple(real)

It would be convenient to allow some form of shorthand for the second case, i.e. allow the tuple aspect of the declaration to be factored out. To do this, an 'exclusive or' operator, @, is introduced. Any object whose type is declared using the @ operator can be typed with either of the operands of @, but once so typed, remains as such for the remainder of its existence. The preceding example could thus be expressed as:

tuple(integer @ real);

Again, this can be used to define a new type mode, and like **generic**, the first value assigned to an object declared as being that mode fixes the type of that mode. In fact, this notion is closely related to that of the mode **generic**, indeed, **generic** can be expressed as:

t1 @ t2 @ t3 ....

for all type symbols in the program's lattice, while **general** can be similarly expressed as:

t1 | t2 | t3 ....

Interestingly enough, this list may be potentially infinite if recursive structures are present in the program.

3.3.5. Typing of Constants

Though the gross data type (i.e. integer, real, etc.) of a constant is obvious, it is desirable to allow the programmer to specify basing information concerning the constants in a program. As an example, given a variable that contains a character string and is declared to be an element of some base, B, if that variable is repeatedly compared with a constant string, being able to specify that the constant is in the base allows equality testing of the two quantities to become nothing more than a pointer equality test, as opposed to a hashing operation followed by a character comparison.

It is possible that the same constant appears more than once in a program, and is to be based differently in each instance. If that is the case some provision must be made for distinguishing between the various instances. The basing information could be placed at the point of occurrence of the constant, but insertion of declarative information into the algorithm proper should be avoided. However, a preferred alternative has not been suggested, and a precedent has already been set in a declaration-free language inserting type information into the program text. B requires a representative denotation to be supplied with every input variable, allowing the translator to deduce the type to be input. Rather than requiring the constant to be typed within the text of the program, we propose that constant that the programmer wishes to be placed in a base be assigned to a variable (in the **init** section of the SETL program) and then declare the variable to be an element of the base.

3.3.6. Sequences and Tuples

The current representation sublanguage has a minor ambiguity in that tuple(integer) could either mean a fixed length tuple of length 1, or an arbitrary length tuple, in both instances the component type being integer. Practically speaking, the point is minor as a tuple of length one is essentially useless. However, a syntactic distinction should be made, if only for the reader, between fixed length tuples, whose purpose is similar to that of a record in Pascal or Ada, and that of a dynamic tuple. The former is often heterogeneous in its component types, while the tuple, with few exceptions, is homogeneous.

Our extended DRSL uses the mode constructor **sequence** to denote fixed length tuples, while **tuple** is reserved for such entities of arbitrary length. For example:

stack : **tuple**(integer | string)

is a declaration of a name which can assume objects that are arbitrary length tuples with integer and string components, while:

symbol_table : **sequence**(string, integer, **set**(string))

declares a fixed length tuple, whose components are a string, integer and set of strings respectively. The question of field specification in sequences is addressed in the next section.

3.3.7. Field Names for Heterogeneous Structures

One cannot directly program a heterogeneous structure in SETL, but must rather resort to using a tuple. The various fields are then accessed using numeric subscripts, and to make this more readable, macros are typically written with names corresponding to the manner in which the components are used. It is more convenient to have the capability to label these fields with their appropriate names when declaring the types of the fields in the DRSL. As an example, in the Ada/Ed compiler the symbol table is maintained as a heterogeneous tuple with several components: name and type of the entity, parameter and discriminant lists, constraint information, etc. The semantic pass, which manipulates this structure defines a series of macros that allows more convenient access to the various fields. A suggested syntax for this capability is:

```
person_info : tuple(name : string,
                    address : tuple(street : string,
                                    city   : string,
                                    state  : string,
                                    zip    : integer))
```

allowing the programmer to then write:

```
name(person_info)    or

street(address(person_info))
```

If field names are forced to be unique within the name scope of macros, then the above facility could be implemented as a special instance of macro processing; that is, the field names could be declared as macros of the following form:

```
macro name(t) =
    t(1)
endm;

macro address(t) =
    t(2)
endm;

macro street(s) =
    s(1)
endm;
```

Of course, implementing field names in the above fashion leaves open much room for programmer error; a particular field name is not constrained to be used exclusively with a variable repr'ed with that 'record' type. A more appropriate implementation would be to incorporate the field naming facility into the representation sublanguage proper. As a field may play more than one role (in addition to assuming more than one type during its lifetime) multiple field names may be associated with a particular component. Introducing discriminants for such 'variant records' is meaningless as the rest of the language is type free.

3.4. Summary

We present in this section a summary of the changes proposed in the above sections. We give a syntax as well as a semantic explanation for each of the new constructs.

```
<declarative section> ::=
 repr
    <declaration;>
    {declaration;}
 end repr;

<declaration> ::=
     <base declaration>
   | <type mode declaration>
   | <object declaration>

<base declaration> ::=
     base <name list> : <optional type>

<type mode declaration> ::=
     mode <name list> : <type>

<object declaration> ::=
     <name list> : <type>

<name list> ::=
     <identifier> {, <identifier>}

<type> ::=
     <single type> {<alternation operator> <single type>}

<single type> ::=
     integer
   | real
   | string
   | atom
   | boolean
   | general
   | *
   | generic
   | set <optional type list>
   | <map type> <optional type list> <optional type>
   | tuple <optional type>
   | sequence <component list>
   | proc <optional parm list> <optional type>
   | op <optional parm list> <optional type>

<alternation operator> ::=
     |
   | @
```

```
<map type> ::=
    mmap
  | smap

<optional type> ::=
    <type>
  | ϵ

<type list> ::=
   <type> {, <type>}

<optional type list> ::=
    ( <type list> )
  | ϵ

<optional parm list> ::=
    ( <optional mode list> <type list> )
  | ϵ

<optional mode list> ::=
    <type mode declaration> , {<type mode declaration> ,}
  | ϵ

<component list> ::=
    ( <component> {, <component> )

<component> ::=
    <optional field name list> <type>

<optional field name list> ::=
    <identifier> {, <identifier>} :
  | ϵ
```

A declarative section is bracketed by the keywords **repr** and **end repr**. The are three forms of declarations: bases, type modes and objects. Bases are declared with an optional type associated with them, if omitted, the base defaults to type **general**. Object declarations allow for the type specification of program objects. Type mode declarations take the same form as object declarations but with the keyword **mode** prefixed to the declaration.

The types are straightforward, * is an abbreviation for **general**, and wherever a type is optional, if omitted, it defaults to **general**. Declaring an object to be **generic** allows it to be initially assigned any type, but its type is then restricted to that type for the remainder of of the scope of the object so declared. Declaring a mode to be **generic** allows for the creation of type-related objects. The first object declared to be of that mode restricts all other objects of that mode to its initial type. The type restriction imposed by the generic mode declaration exists for the lifetime of the procedure within which the mode is declared.

The exclusive alternation operator acts in the same fashion as type **generic**, in that it restricts the type of the object to it initial type. In addition, it allows for a restricted set of possible types to be specified.

CHAPTER 4

A SETL Implementation of the Typefinder

4. A SETL Implementation of the Typefinder

4.1. Introductory Remarks

The following program is an implementation of the typefinder presented in this thesis. It is a standalone program, not interfaced with the existing SETL optimizer and therefore several simplifications were made. The **read** statement is handled as a primitive operation as opposed to a built-in function. The input to the typefinder is the original SETL source of a program which is read in as a comment, followed by the intermediate operations which have been hand-coded. The UD chains are also supplied to the program as data. In a working version incorporated into the optimizer, all this information would be provided by the front end.

4.2. The Algorithm

```
program type_finder;

$ The following is a prototype implementation of a typefinder capable of
$ detecting recursive data structures.

$ Typefinding works in the following fashion:
$
$ An initialization pass is performed over the code. This pass types any
$ constants, places in the workpile any ovariables that have constant right hand
$ sides and assigns recursive type names to any nonconvergent ivariables.
$ It also tests the input variables of any embedding operation (e.g. Q1_TUP)
$ for recursiveness (i.e. whether the definition being constructed reaches
$ the input variable). Each new (recursive) type is mapped via TYPE_STRUCTURE to
$ a type representing its structure - which is initially set to TYPE_ERROR. The
$ TYPE_OF map for any variable flagged as nonconvergent is set to the recrusive
$ name it is assigned.
$
$ Forward analysis proceeds by removing an element from the workpile. If the
$ element is an ivariable, the types of all definitions reaching that ivariable
$ are 'joined' and the result is placed in TYPE_OF if the ivariable is not
$ nonconvergent and in TYPE_STRUCTURE otherwise. For ovariables, the procedure
$ forward is invoked which calculates the result type based upon the input
$ types. The type used for nonconvergent ivariables here will simply be their
$ type names. The exception to this rule is when a recursive type
$ which is an input argument must have its component types
$ examined (e.g. for a tuple or set extraction). In that
$ case, the structure is unfolded one level to allow the appropriate component
$ to be accessed.
$
$ After the occurrence has been processed, if any change has occurred (to
$ TYPE_OF for ovariables and ivariables that are not nonconvergent, or to
$ TYPE_STRUCTURE for nonconvergent ivariables), affected occurrences are
$ dumped into the workpile.

$ The above process continues until the workpile is empty.


$ Types
$ =====
$
$ The version of the typefinder that detects recursive data structures maintains
$ more precise information concerning types of variable instances than its
$ predecessor. As such, the data structure used to contain type information is
$ more detailed..

$ The primitive types are: boolean, real, integer, string, atom, the nullset,
$ the nulltuple, set, seq (known length tuple), and tuple ('homogeneous' tuple).

$ Each elementary type symbol (i.e. non alternated) is represented by the
$ format:
$
```

```
$      gross_type                      for scalar types
$      [gross-type, component-type]    for set, tuple and seq
$
$ The component type is itself a type symbol and thus a recursive walk is
$ required to analyze the full type symbol. If the gross-type is a sequence, the
$ component type is a tuple with the i'th component representing the type symbol
$ of the i'th element of the sequence.
$
$
$ Examples:
$  integer                        - integer
$  [set, integer]                 - set(integer)
$  [tuple, integer]               - tuple(integer)
$  [seq, [integer, string]]       - tuple(integer, string)
$  [set, [seq, [string, integer]]]  - map(string) integer
$

$ An type symbol is maintained as a set containing the elementary type symbols
$ that compose its alternands.
$
$ Example:
$  {integer}                         - integer
$  {integer, string}                 - integer | string
$  {integer [set, integer]}          - integer | set(integer)
$

$ Disjunction operator
$
$ The disjunction (meet, union) operator accepts two type symbols and produces
$ a third. To maintain compactness of the type symbols, the following rules
$ hold:

$ - a type symbol may have only one SET elementary type. The result of
$   disjuncting two symbols with SET alternands is a SET with a component type
$   consisting of the disjunction of the two original component types.
$
$ - The above also applies for TUPLE elementary types.
$
$ - If a symbol contains a TUPLE elementary type, it should not contain a SEQ
$   type. Thus, if two symbols are disjuncted and one contains a TUPLE type,
$   the SEQ types in the other are compressed (i.e. their positional component
$   types are disjuncted), disjuncted with the component type of the TUPLE,
$   and the resulting type is used as the component type of a TUPLE.
$
$ - SEQ types are merged if their component types are of the same size.


macro TYPES;
  T_BOOLEAN,
  T_ATOM,
  T_REAL,
  T_INTEGER,
  T_STRING,
   T_NULLSET,
   T_NULLTUP,
   T_SET,
   T_TUPLE,
   T_SEQ,
   T_GENERAL,
   T_ERROR,
   T_OM
endm;


$ Macros used in type information extraction


$ Will return true for everything except TUPLE, SEQ and SET

macro is_scalar_type(t);
  not is_tuple t
endm;

$ Returns type for scalar type, T_TUP, T_SEQ, or T_SET otherwise

macro gross_type(t);
  (if is_scalar_type(t) then
      t
   else
      t(1)
   end)
endm;

$ For compound types returns component types.

macro component_type(t);
  (if t = T_GENERAL then    $ TYPE_GENERAL has component type of TYPE_GENERA
      TYPE_GENERAL
   else
      t(2)
   end)
endm;

macro number_of_alternands(t);
  #(t)
endm;


$ Data Structures
$
$ The algorithm assumes the existence of several data structures already
$  calculated by the SETL optimizer (though in possibly different formats).
$  These are:
$
$  UD   : multivalued map from a variable use to the set of instances which
$                    define that use.
$ DU   : inverse map of ud - calculated from UD.
$

$ Instruction Format
$
$ The program is represented by a set of instructions each tagged with a unique
$ integer. As ud information is assumed, and the algorithm is workpile based,
$ no other relationships regarding the instructions (such as basic block
$ membership or next instruction) need be maintained.
$ Each instruction represented by the maps OPCODE and ARGS. The output variable
$ of the instruction is the first of the arguments.
$ With the exception of the instruction's argument list, all variables are
$ represented as occurrences, i.e. a tuple whose first element is the
$ instruction number and whose second element is the position of the variable
$ in the argument list.


$ Macros for accessing instructions and occurrences.

macro ovar(i);        [i, 1]                        endm;
macro ivars(i);       [[i, j] :
                         j in [2 .. #ARGS(i)]]      endm;
macro ivar1(i);       [i, 2]                        endm;
macro ivar2(i);       [i, 3]                        endm;
macro ivar(i,n);      [i, n+1]                      endm;

macro is_ivar(occ);    occ(2) > 1                   endm;
macro is_ovar(occ);   occ(2) = 1                    endm;

macro arg_num(occ);    occ(2)                       endm;
macro name_of(occ);   ARGS(occ(1))(occ(2))          endm;
macro instr_of(occ);    occ(1)                      endm;

macro is_constant(occ);
  constant_type(occ) /= TYPE_ERROR
endm;


$ Set of Q1 Opcodes

macro OPCODES;

$ Binary Operators

   Q1_ADD,            $ +
   Q1_DIV,            $ div
   Q1_EXP,            $ **
   Q1_EQ,             $ =
   Q1_GE,             $ >=
   Q1_LT,             $ <
   Q1_POS,            $ >0
   Q1_IN,             $ in
   Q1_INCS,           $ incs, subset
```

```
    Q1_LESS,            $ less
    Q1_LESSF,           $ lessf
    Q1_MAX,             $ max
    Q1_MIN,             $ min
    Q1_MOD,             $ //
    Q1_MULT,            $ *
    Q1_NE,              $ /=
    Q1_NOTIN,           $ notin
    Q1_NPOW,            $ npow
    Q1_ATAN2,           $ atan2
    Q1_SLASH,           $ /
    Q1_SUB,             $ -
    Q1_WITH,            $ with

$ Unary Operators

    Q1_ABS,             $ abs
    Q1_CHAR,            $ char
    Q1_CEIL,            $ ceiling
    Q1_FLOOR,           $ floor
    Q1_ISINT,           $ is_integer
    Q1_ISREAL,          $ is_real
    Q1_ISSTR,           $ is_string
    Q1_ISBOOL,          $ is_boolean
    Q1_ISATOM,          $ is_atom
    Q1_ISTUP,           $ is_tuple
    Q1_ISSET,           $ is_set
    Q1_ISMAP,           $ is_map
    Q1_ARB,             $ arb
    Q1_VAL,             $ val
    Q1_DOM,             $ domain
    Q1_FIX,             $ fix
    Q1_FLOAT,           $ float
    Q1_NELT,            $ nelt
    Q1_NOT,             $ not
    Q1_POW,             $ pow
    Q1_RAND,            $ random
    Q1_SIN,             $ sin
    Q1_COS,             $ cos
    Q1_TAN,             $ tan
    Q1_ARCSIN,          $ asin
    Q1_ARCCOS,          $ acos
    Q1_ARCTAN,          $ atan
    Q1_TANH,            $ tanh
    Q1_EXPF,            $ exp
    Q1_LOG,             $ log
    Q1_SQRT,            $ sqrt
    Q1_RANGE,           $ range
    Q1_TYPE,            $ type
    Q1_UMIN,            $ unary minus
    Q1_EVEN,            $ even
    Q1_ODD,             $ odd
    Q1_STR,             $ str
```

```
  Q1_SIGN,          $ sign

$ Miscellaneous

  Q1_END,           $ A1 := A2(A3 ..)
  Q1_SUBST,         $ A1 := A2(A3 .. A4)
  Q1_NEWAT,         $ A1 := newat
  Q1_TIME,          $ A1 := time
  Q1_DATE,          $ A1 := date
  Q1_NA,            $ A1 := # of arguments of current routine
  Q1_SET,           $ enumerative set former
  Q1_SET1,          $ iterative set former
  Q1_TUP,           $ enumerative tuple former
  Q1_TUP1,          $ iterative tuple former
  Q1_FROM,          $ A1 from A2
  Q1_FROMB,         $ A1 fromb A2
  Q1_FROME,         $ A1 frome A2

$ Iterators

  Q1_NEXT,          $ A1 := next element of A3
  Q1_NEXTD,         $ A1 := next element of domain A3
  Q1_INEXT,         $ initialize next loop
  Q1_INEXTD,        $ initialize nextd loop

$ Mappings

  Q1_OF,            $ A1 := A2(A3)
  Q1_OFA,           $ A1 := A2{A3}
  Q1_SOF,           $ A1(A2) := A3
  Q1_SOFA,          $ A1{A2} := A3
  Q1_SEND,          $ A1(A2 ..) := A3
  Q1_SSUBST,        $ A1(A2 .. A3) := A4

$ Assignments

  Q1_ASN,           $ A1 := A2
  Q1_ARGIN,         $ assign argument to formal parameter
  Q1_ARGOUT,        $ assign back to argument
  Q1_PUSH,          $ push element for set former
  Q1_READ,          $ kludge for read

endm;


var
 OPCODE,                  $ Instruction's opcode
 ARGS,                    $     "        arguments
 TYPE_STRUCTURE,          $ Structure of discovered recursive types
 REPRESENTATIVE_TYPE.     $ Once types are merged, reprsentative type of class
 UD,                      $ Ud chains
 DU,                      $ Du chains
 INSTRUCTIONS,            $ tuple of instructions
 TYPE_OF,                    $ map of types for each occurrence
 VAR_TYPES,                  $ types for each variable
 workpile,                   $
 is_rec_type,                $ Boolean map for testing is a type is recursive
 is_recursive,               $ Boolean map for recursive ivars
 occ_depends_upon;


const
  OPCODES,
  TYPES,
  TYPE_BOOLEAN,
  TYPE_ATOM,
  TYPE_REAL,
  TYPE_INTEGER,
  TYPE_STRING,
  TYPE_NULLSET,
  TYPE_NULLTUP,
  TYPE_GENERAL,
  TYPE_ERROR,
  TYPE_OM,

  $ Operators that increase nesting level

  EMBEDDING_OPS = {Q1_TUP, Q1_SET, Q1_WITH, Q1_SOF};


$ Initialize global structures

TYPE_OF      := {};                  $ Type symbol for each occ
VAR_TYPES := {};                     $ Final type symbol for each variable
TYPE_STRUCTURE := {};                $ Internal type stucture for recursive types
REPRESENTATIVE_TYPE := {};           $ representative recursive type for merge phase

OPCODE  := [];                       $ Tuple of programs Q1 opcodes
ARGS   := [];                        $ Tuple of Q1 arguments

is_rec_type           := {};  $ Will be set to true for recursive type symbols
is_recursive          := {};  $ True for occs that are recursive (see intro)
occ_depends_upon      := {};  $ For each recursive type, the set of ovars
                              $  dependent upon structure of that type


$ Read in assumed structures

$ Read in program description: Original source code is read in as a comment
$  for clarity

print ('Original source code:');
print (' ');

get ('', stmt);
(while stmt /= '/*')
```

```
  print (stmt + ' ');
  get ('', stmt);
end while;

eject ();

$ Read in instruction tuple and extract various fields

read (INSTRUCTIONS);

(for inst = INSTRUCTIONS(i))

   $ Extract fields of the instruction

   [-, opcd, arguments] := inst;
   OPCODE(i) := opcd;
   ARGS(i)   := arguments;

end for inst;

read (UD);

(for u in domain UD)

  $ Make sure no errors in input

  ASSERT is_ivar(u);

  (for d in UD{u})
    ASSERT is_ovar(d);
    ASSERT name_of(u) = name_of(d);
  end for d;

end for u;


DU := {[d,u] : [u,d] in UD};      $ compute DU = inverse UD

initialize_workpile;              $ Type constants, find recursive occurrences and
                                  $  place definitions on workpile

eject ();


$ Actual forward analysis

forward_analysis;

$ Merge isomorphic types

merge_recursive_types;

$ Merge instances of the same variable
```

```
merge_instances;

print_results;

stop;


proc print_results;

eject ();

print ('Recursive Type Structures');
print (' ');

(for r in domain TYPE_STRUCTURE)
   print (r, ' = ');
   pretty_print(TYPE_STRUCTURE(r));
   end for r;

print (' ');
print (' ');
print ('Types');
print (' ');

(for v in domain VAR_TYPES)
   print (v, ' : ');
   pretty_print (VAR_TYPES(v));
end for v;

return;

end proc print_results;


proc initialize_workpile;

$ initialize_workpile assigns types to all constant value and places all
$ definitions onto the workpile.

workpile := {};

$ Loop through all instructions

(for i in [1 .. #INSTRUCTIONS])
   const_rhs := true;

   $ Process ivars of instruction

   (for a in ivars(i))
      TYPE_OF(a) := constant_type(a);          $ Type any constants
      if TYPE_OF(a) = TYPE_ERROR then          $ Not constant?
         const_rhs := false;
      end if;
```

```
  $ If the opcode of the current instruction results in an embedding,
  $ test the ivariable for recursiveness. A variable is recursive if
  $ the definition it is building reaches it.

     if OPCODE(i) in EMBEDDING_OPS then
        is_recursive(a) := test_for_recursiveness(a);
        if is_recursive(a) then
           new_type := generate_new_type();        $ new recursive type symbol
           TYPE_OF(a) := new_type;
           TYPE_STRUCTURE(new_type) := TYPE_ERROR;
           is_rec_type(new_type) := true;          $ Flag the type as recursive
        end if;
     end if;
   end for a;

   $ Only place into the workpile those ovars that have a constant right hand
   $  side.

   if const_rhs then
      workpile with:= ovar(i);
   end if;

   TYPE_OF(ovar(i)) := TYPE_ERROR;

end for i;

end proc initialize_workpile;


proc test_for_recursiveness(instance);

$ To test for recursiveness, see if the instance can reach itself. To do this,
$ we use a workpile which is initialized to the instance. When an item is
$ removed from the pile, if it is an ivar, the corresponding ovar is placed into
$ the pile; if an ovar, all ivars reachable by that ovar are placed into the
$ pile. Furthermore, the removed item is remembered so as not to process it a
$ second time (in the event of other recursive instances on the path).
$ If the original instance ever appears in the pile, it is recursive.

pile := {instance};
seen := {};

(while pile /= {})

   occ from pile;
   seen with:= occ;          $ Remember that this occurrence has been prcessed
                             $  to prevent infinite loops.

   $ Ivars send their corresponding ovars onto the pile; ovars send
   $ all subsequent uses into the pile.

   if is_ivar(occ) then
      pile with:= ovar(instr_of(occ));
```

```
    else
      pile +:= {use : use in DU{occ}};
      if instance in pile then                   $ Was occurrence in question placed into
        return true;                             $  pile? - if so it's recursive.
      end if;
    end if;

    pile -:= seen;             $ Remove any processed occurrences that
                                 $  may have just been placed onto pile.
  end while pile;

  return false;                $ original occurrence never appeared

  end proc test_for_recursiveness;


  proc forward_analysis;

  $ Actual routine to perform forward analysis. Items are removed from the
  $ workpile, processed in an appropriate fashion, and all affected items are
  $ inserted into the pile.


  (while workpile /= {})

    occ from workpile;             $ Remove an occurrence from the workpile


    $ For nonconvergent ivariables, check for change in TYPE_STRUCTURE, otherwise
    $  change occurs in TYPE_OF

    if is_recursive(occ) ? false and
      is_ivar(occ) then
      oldtype := TYPE_STRUCTURE(TYPE_OF(occ));
    else
      oldtype := TYPE_OF(occ);                 $ Retain its previous type
    end if;

    $ Ovars have their types calculated via forward; ivars by taking the
    $  disjunction of all chained definitions

    if is_ovar(occ) then                           $ occurrence is an ovar
      TYPE_OF(occ) := forward(instr_of(occ));
    else                                           $ occurrence is an ivar
      ASSERT is_ivar(occ);

      $ for nonconvergent occurrences, 'join' is placed in TYPE_STRUCTURE

      if is_recursive(occ) ? false then
        (for d in UD{occ})
          TYPE_STRUCTURE(TYPE_OF(occ)) .DIS:=
                TYPE_OF(d);
        end for;
      TYPE_STRUCTURE(TYPE_OF(occ)) -:= TYPE_OF(occ);
    else
      TYPE_OF(occ) := TYPE_ERROR .DIS/
                       [TYPE_OF(d) : d in UD{occ}];
    end if;
  end if;


  $ Check for change in type of processed occurrence. This change will
  $  occur in TYPE_OF for nonrecursive variables and TYPE_STRUCTURE
  $  otherwise.

  if is_recursive(occ) ? false and
    is_ivar(occ) then
    if oldtype = TYPE_STRUCTURE(TYPE_OF(occ)) then
      continue;
    end if;
  elseif oldtype = TYPE_OF(occ) then
    continue;
  end if;


  $ Ovariables dump their immediate uses into the workpile;
  $  ivars send their corresponding definition together with the
  $  image of occ_depends_upon for the type of the occurrence if
  $  recursive.

  if is_ovar(occ) then                          $ modified ovar
    (forall use in DU{occ} ? {})
      workpile with:= use;
    end forall;
  else                                          $ modified ivar
    ASSERT is_ivar(occ);
    o := ovar(instr_of(occ));
    workpile with:= o;
    if is_rec_type(TYPE_OF(occ)) ? false then
      workpile +:= occ_depends_upon{TYPE_OF(occ)} ? {};
    end if;
  end if;

end while;

end proc forward_analysis;


proc constant_type(occ);

$ Examine an occurrence; if it is a constant, return its type else return
$ TYPE_ERROR as an initial value for variables.

name := name_of(occ);                   $ Extract identifier name of occurrence

if  name = 'OM' then                          $ om
```

```
  result := TYPE_OM;
elseif name = 'NULLSET' then                $ null set
  result := TYPE_NULLSET;
elseif name = 'NULLTUP' then                $ null tuple
  result := TYPE_NULLTUP;
elseif is_integer(name) then            $ integer constant
  result := TYPE_INTEGER;
elseif is_string(name) and              $ string constant
  name(1) = '''' then
  result := TYPE_STRING;
else                                    $ variable - not a constant
  result := TYPE_ERROR;
end if;

return result;

end proc constant_type;


proc generate_new_type;

$ Generate new recursive type symbol of form REC_xxx

new_type := {'REC_' + str newat};

return new_type;

end proc generate_new_type;


proc forward(inst);

$ Generate result type of output variable based upon operation and types of
$  input variables

i1 := ivar1(inst);
i2 := ivar2(inst);

result := TYPE_ERROR;               $ Default type

case OPCODE(inst) of

$ Binary Operators

  (Q1_IN, Q1_NOTIN):

    $ if T2 is a set or tuple, the result is BOOLEAN

    t2 := collect_types(i2);
    if t2 .INTER {T_SET, T_TUPLE, T_SEQ, T_STRING} /= {} then
      result := TYPE_BOOLEAN;
    end if;
```

```
(Q1_INCS):

  $ if both inputs are sets, the result is BOOLEAN

  t1 := collect_types(i1);
  t2 := collect_types(i2);
  if t1 .INTER {T_SET} /= {} and
     t2 .INTER {T_SET} /= {} then
     result := TYPE_BOOLEAN;
  end if;

(Q1_EQ,Q1_NE):

  $ Always returns Boolean

  result := TYPE_BOOLEAN;

(Q1_GE, Q1_LT, Q1_POS):

  $ If operands are valid, returns Boolean

  t1 := collect_types(i1);
  t2 := collect_types(i2);
  temp1 := t1 .INTER {T_INTEGER, T_REAL, T_STRING};
  temp2 := t2 .INTER {gross_type(t) : t in temp1};
  if temp2 /= {} then
     result := TYPE_BOOLEAN;
  end if;

(Q1_WITH):

  $ If t1 is a set or tuple, then its type is the gross type
  $ with a component type consisting of the disjunction of the component type
  $ of t1 and t2. Known length tuples just tack t2 onto the back of the
  $ component type of t1.

  t1 := collect_types(i1);
  t2 := TYPE_OF(i2);
  if (typ := t1 .INTER {T_SET, T_TUPLE}) /= {} then
     result := {[gross_type(etyp), component_type(etyp) .DIS t2] :
                   etyp in typ};
  end if;
  if (typ := t1 .INTER {T_SEQ}) /= {} then

                 $ if the operand is recursive, transform a known-length
                 $ sequence into a tuple

     if is_rec_type(TYPE_OF(i1)) ? false then
        result .DIS:= {[T_TUPLE, t2 .DIS/{.DIS/component_type(etyp) :
                          etyp in typ}]};
     else
        result .DIS:= {[T_SEQ, component_type(etyp) with t2] :
                   etyp in typ};
```

```
    end if;
  end if;

        $ If first arg is a null set or tuple, create set or tuple type

  if t2 /= TYPE_ERROR then
    if (typ := t1 .INTER {T_NULLTUP}) /= {} then
      result .DIS:= {[T_SEQ, [t2]]};
    end if;
    if (typ := t1 .INTER {T_NULLSET}) /= {} then
      result .DIS:= {[T_SET, t2]};
    end if;
  end if;

(Q1_LESS):

  $ if t1 is a set, the result is same as t1

  t1 := collect_types(i1);
  if (temp := t1 .INTER {T_SET}) /= {} then
    result := TYPE_NULLSET .DIS {[T_SET, .DIS/ {component_type(t) :
                                           t in temp}]};
  end if;

(Q1_NPOW):

  $ One input is a set , the other an integer, the result is
  $ a set of type t1.

  t1 := collect_types(i1);
  t2 := collect_types(t2);
  if t1 .INTER {T_SET} /= {} and
    t2 .INTER {T_INTEGER} /= {} then
    result := {[T_SET, t1]};
  end if;
  if t2 .INTER {T_SET} /= {} and
    t1 .INTER {T_INTEGER} /= {} then
    result := {[T_SET, t2]};
  end if;

(Q1_MAX, Q1_MIN):

  t1 := collect_types(i1);
  t2 := collect_types(i2);
  temp1 := t1 .INTER {T_INTEGER, T_REAL, T_STRING};
  temp2 := t2 .INTER {gross_type(t) : t in temp1};
  if temp2 /= {} then
    result := temp2;
  end if;

(Q1_ADD):

  t1 := collect_types(i1);
  t2 := collect_types(i2);

  $ Scalars are intersected

  temp1 := t1 .INTER {T_INTEGER, T_REAL, T_STRING};
  temp2 := t2 .INTER {gross_type(t) : t in temp1};
  if temp2 /= {} then
    result := temp2;
  end if;

  $ Sets have their component types disjuncted

  temp1 := t1 .INTER {T_SET};
  temp2 := t2 .INTER {T_SET};
  if temp1 /= {} and
    temp2 /= {} then
    result .DIS:= {[T_SET, .DIS/{component_type(t) :
                          t in temp1} .DIS
                      .DIS/{component_type(t) :
                          t in temp2}]};
  end if;

  $ Tuples and sequences are transformed into tuples with
  $ disjuncted component types

  temp1 := t1 .INTER {T_SEQ, T_TUPLE};
  temp2 := t2 .INTER {T_SEQ, T_TUPLE};
  if temp1 /= {} and
    temp2 /= {} then
    result .DIS:= {[T_TUPLE, .DIS/{component_type(t) :
                          t in temp1 |
                          gross_type(t) = T_TUPLE} .DIS
                        .DIS/{.DIS/component_type(t) :
                          t in temp1 |
                            gross_type(t) = TSEQ}  .DIS
                        .DIS/{component_type(t) :
                          t in temp2 |
                          gross_type(t) = T_TUPLE} .DIS
                        .DIS/{.DIS/component_type(t) :
                          t in temp2 |
                            gross_type(t) = TSEQ}]};
  end if;

(Q1_SUB):
  t1 := collect_types(i1);
  t2 := collect_types(i2);
  temp1 := t1 .INTER {T_INTEGER, T_REAL};
  temp2 := t2 .INTER {gross_type(t) : t in temp1};
  if temp2 /= {} then
    result := temp2;
  end if;
  if (typ := t1 .INTER {T_SET}) /= {} and
    t2 .INTER {T_SET} /= TYPE_ERROR then
    result .DIS:= typ;
  end if;

 (Q1_DIV):
  t1 := collect_types(i1);
  t2 := collect_types(i2);
  if t1 .INTER {T_INTEGER} /= {} and
    t2 .INTER {T_INTEGER} /= {} then
    result := TYPE_INTEGER;
  end if;

 (Q1_SLASH):
  t1 := collect_types(i1);
  t2 := collect_types(i2);
  temp1 := t1 .INTER {T_INTEGER, T_REAL};
  temp2 := t2 .INTER {gross_type(t) : t in temp1};
  if temp2 /= {} then
    result := TYPE_REAL;
  end if;

 (Q1_EXP):
  t1 := collect_types(i1);
  t2 := collect_types(i2);
  temp1 := t1 .INTER {T_INTEGER, T_REAL};
  temp2 := t2 .INTER {gross_type(t) : t in temp1};
  if temp2 /= {} then
    result := temp2;
  end if;
  if t1 .INTER {T_REAL} /= {} and
    t2 .INTER {T_INTEGER} /= {} then
    result .DIS:= TYPE_REAL;
  end if;

 (Q1_ATAN2):
  t1 := collect_types(i1);
  t2 := collect_types(i2);
  if t1 .INTER {T_REAL} /= {} and
    t2 .INTER {T_REAL} /= {} then
    result .DIS:= TYPE_REAL;
  end if;


$ Unary operators

 (Q1_NOT):
  t1 := collect_types(i1);
  result := t1 .INTER {T_BOOLEAN};

 (Q1_EVEN, Q1_ODD):
  t1 := collect_types(i1);
  if t1 .INTER {T_INTEGER} /= {} then
    result := TYPE_BOOLEAN;
  end if;
  (Q1_ISINT, Q1_ISREAL, Q1_ISSTR, Q1_ISBOOL,
   Q1_ISATOM, Q1_ISTUP, Q1_ISSET, Q1_ISMAP):
   t1 := collect_types(i1);
   result := TYPE_BOOLEAN;

  (Q1_ARB):

   $ if t1 is a set then the result type is the component type

   t1 := collect_types(i1);
   if (typ := t1 .INTER {T_SET}) /= {} then
           typ := arb typ;
     result := component_type(typ);
   end if;

  (Q1_POW):
   t1 := collect_types(i1);
   if (typ := t1 .INTER {T_SET}) /= {} then
     result := {[T_SET, [t1]]};
   end if;

  (Q1_NELT):
   t1 := collect_types(i1);
   if (typ := t1 .INTER {T_SET, T_TUP, T_STRING}) /= {} then
     result := {[T_SET, [t1]]};
   end if;


$ Assigning Operators

  (Q1_ASN):
   result := TYPE_OF(i1);


$ Map and Tuple Operations

  (Q1_OF):
   t1 := collect_types(i1);
   (for etyp in t1)
       if gross_type(etyp) = TYPE_STRING and
         t2 .INTER {T_INTEGER} /= {} then
         result .DIS:= TYPE_STRING;
       end if;

       $ If tuple, disjunct in all component types

       if gross_type(etyp) = T_TUPLE then
         result .DIS:= component_type(etyp);

       $ If sequence, then if subscript is constant
       $  use the corresponding component type, otherwise
       $  take disjunction of all component types
```

```
        elseif gross_type(etyp) = T_SEQ then
           if is_constant(ivar2(inst)) then
              result .DIS:=
                  component_type(etyp)(name_of(ivar2(inst))) ?
                           TYPE_OM;
           else
              result := result .DIS/ component_type(etyp);
           end if;

        $ Check for map access

        elseif gross_type(etyp) = T_SET then
           (for comptyp in component_type(etype))
                if gross_type(comptyp) = T_SEQ and      $ have a pair
                   #(ctyp := component_type(comptyp)) = 2 then
                   temp1 := ctyp .INTER {gross_type(t) :
                             t in t2};
                   if temp1 /= {} then
                      result .DIS:= ctyp(2);
                   else
                      result .DIS:= TYPE_OM;
                   end if;
                end if;
            end for;
        end if;
    end for;

  (Q1_SOF):

    $ We handle only cases of tuples and sequences

    t1 := collect_types(i1);
    (for etyp in t1)

       $ If tuple, merge in type of source operand

       if gross_type(etyp) = T_TUPLE then
          result .DIS:= {[T_TUPLE, component_type(etyp) .DIS
                      TYPE_OF(ivar(inst,3))]};
       end if;

       $ If sequence, then if constant subscript, update
       $  component type tuple, else merge all component types into
       $  one and set gross type to tuple

       if gross_type(etyp) = T_SEQ then
          if is_constant(ivar2(inst)) then
             if 1 <= name_of(ivar2(inst)) and
                name_of(ivar2(inst)) <= #component_type(etyp) then
                compres := component_type(etyp);
                compres(name_of(ivar2(inst))) .DIS:=
                     TYPE_OF(ivar(inst,3));
                result .DIS:= {[T_SEQ, compres]};
          elseif name_of(ivar2(inst)) >=
                                 #component_type(etyp) then
            compres := component_type(etyp);
            (for i in [#component_type(etyp) ..
                                    name_of(ivar2(inst))-1])
                compres(i) := TYPE_OM;
            end for i;
            compres(name_of(ivar2(inst))) :=
                                        TYPE_OF(ivar(inst,3));
            result .DIS:= {[T_SEQ, compres]};
          end if 1;
        else                    $ Not constant subscript
          result .DIS:= {[T_TUPLE, .DIS/component_type(etyp)]};
        end if is_constant(ivar2(inst));
     end if gross_type(etyp) = T_SEQ;
  end for etyp;

(Q1_SET):
      component := TYPE_ERROR;
      valid := true;
      (for i in [1 .. #ivars(inst)])
        elemtype := TYPE_ERROR .DIS TYPE_OF(ivar(inst,i));
        if elemtype = TYPE_ERROR then
           valid := false;
           quit;
        else
           component .DIS:= elemtype;
        end if;
      end for i;

      $ Only form set if all components had non error type

      if valid then
         result := {[T_SET, component]};
      else
         result := TYPE_ERROR;
      end if;

(Q1_TUP):
   components := [];
   valid := true;
   (for i in [1 .. #ivars(inst)])
       elemtype := TYPE_ERROR .DIS TYPE_OF(ivar(inst,i));
       if elemtype = TYPE_ERROR then
          valid := false;
          quit;
       else
          components with:= elemtype;
       end if;
   end for i;

   $ Only form tuple type if all components are not TYPE_ERROR
```

```
    if valid then
      result := {[T_SEQ, components]};
    else
      result := TYPE_ERROR;
    end if;

  (Q1_READ):

  $ *** Kludge for read, since we're not handling built-in functions.

    result := TYPE_GENERAL;

end case;

return result;

end proc forward;


proc collect_types(occ);

$ collect_types examines the type structure of an ivariable for use by the
$ forward propagation functions. If the variable is nonrecursive it examines the
$ TYPE_OF map, otherwise its looks at the TYPE_STRUCTURE map.

$ collect types also updates the occ_depends_on map which determines which
$ occurrences are affected by changes to the TYPE_STRUCTURE of a recursive type.

ASSERT is_ivar(occ);

typ := TYPE_OF(occ);
inst := instr_of(occ);

$ if recursive ivariable, get types from its TYPE_STRUCTURE

if is_rec_type(typ) ? false then
    occ_depends_upon{typ} := occ_depends_upon{typ} ? {} with ovar(inst);
    typ := TYPE_STRUCTURE(typ);
    types_seen := typ;
else
    types_seen := {};
end if;

$ If there is in the set of types collected a recursive symbol, remove
$ it and replace it with its TYPE_STRUCTURE

(while exists etyp in typ | (is_rec_type({etyp}) ? false))
   types_seen with:= etyp;
   typ .DIS:= TYPE_STRUCTURE({etyp});
   typ -:= types_seen;
   occ_depends_upon{{etyp}} := occ_depends_upon{{etyp}} ? {} with
         ovar(inst);
end while;
```

```
return typ;

end proc collect_types;


proc merge_recursive_types;

$ merge recursive types that are name equivalent into classes. To do this
$ first bring in the TYPE_STRUCTURE of any recursive type name at the top level
$ of TYPE_STRUCTURE. Then use the TYPE_STRUCTURE values as criteria for
$ two types being equal. If they are, replace one type name by the
$ representative type of that class.

(for rectyp in domain TYPE_STRUCTURE)

    $ types_seen is used to ensure that the bringing in of TYPE_STRUCTURES
    $ does not enter a cycle. Since the number of recursive type names are
    $ finite, and we are constantly removing already processed types from the
    $ set of candidates, the loop will eventually terminate

    types_seen := {};
    (while exists typ in TYPE_STRUCTURE(rectyp) |
        is_rec_type({typ}) ? false)
        types_seen with:= typ;
        if {typ} /= rectyp then
            TYPE_STRUCTURE(rectyp) .DIS:= TYPE_STRUCTURE({typ});
        end if;
        TYPE_STRUCTURE(rectyp) -:= types_seen;
    end while exists typ;
end for rectyp;

(for rectyp in domain TYPE_STRUCTURE)
    if REPRESENTATIVE_TYPE(TYPE_STRUCTURE(rectyp)) = om then
        REPRESENTATIVE_TYPE(TYPE_STRUCTURE(rectyp)) := rectyp;
    else
        TYPE_STRUCTURE(rectyp) :=
                    REPRESENTATIVE_TYPE(TYPE_STRUCTURE(rectyp));
    end if;
end for rectyp;

(for typ in domain TYPE_STRUCTURE |
    not (is_rec_type(TYPE_STRUCTURE(typ)) ? false))
    TYPE_STRUCTURE(typ) :=
        substitute_recursive_types(TYPE_STRUCTURE(typ));
end for typ;

(for typ in domain TYPE_OF)
        TYPE_OF(typ) := substitute_recursive_types(TYPE_OF(typ));
end for typ;

end proc merge_recursive_types;
```

```
proc merge_instances;

$ After type processing, the types of all instances of the same variable are
$  disjuncted into a single type symbol for the variable

(for i in [1 .. #INSTRUCTIONS])
  VAR_TYPES(name_of(ovar(i))) := VAR_TYPES(name_of(ovar(i))) ?
                              TYPE_ERROR .DIS TYPE_OF(ovar(i));
  (for a in ivars(i))
    VAR_TYPES(name_of(a)) := VAR_TYPES(name_of(a)) ? TYPE_ERROR .DIS
                              TYPE_OF(a);
  end for a;
end for i;

return;

end proc merge_instances;


proc substitute_recursive_types(typ);

$ replace occurrences of recursive type symbols in the resulting type
$  equations by their representative type symbols

result := {};

(for etyp in typ)
  if is_rec_type({etyp}) ? false then
    if is_rec_type(TYPE_STRUCTURE({etyp})) ? false then
       result +:= TYPE_STRUCTURE({etyp});
    else
       result with:= etyp;
    end if;
  elseif is_scalar_type(etyp) then
    result with:= etyp;
  else
    if gross_type(etyp) in {T_SET, T_TUPLE} then
       ctyp := component_type(etyp);
       result with:= [gross_type(etyp),
                      substitute_recursive_types(ctyp)];
    else
       ASSERT gross_type(etyp) = T_SEQ;
       ctyp := component_type(etyp);
       result with:= [T_SEQ, [substitute_recursive_types(ctyp(i)) :
                         i in [1 .. #ctyp]]];
    end if;
  end if;
end for etyp;

return result;

end proc substitute_recursive_types;
```

```
op .DIS(type1, type2);

$ Disjunction operator

if type1 = type2 then                          $ types are equal
  result := type1;
elseif type1 = TYPE_ERROR then              $ dis(error, t) = dis(t, error) = t
  result := type2;
elseif type2 = TYPE_ERROR then
  result := type1;
elseif type1 = TYPE_GENERAL or              $ dis(general, t) = dis(t, general) =
     type2 = TYPE_GENERAL then              $  general
  result := TYPE_GENERAL;

$ Attempt to merge recursive and nonrecursive types
$ If a nonrecursive type is a subset of the TYPE_STRUCTURE of the
$ second type, the result is the second type, unless the two are equal,
$ in which case a normal disjunction is performed. This last condition
$ is to prevent the following:
$ Say we have REC_#1 : INTEGER  disjuncted with INTEGER.
$ If the merge takes place, the result type is REC_#1. If this result is
$ to be placed back into the type structure of REC_#1, the leaf type will
$ never be produced.

elseif (is_rec_type(type1) ? false and
  not (is_rec_type(type2) ? false)) and
  (type2 subset TYPE_STRUCTURE(type1) and
      type2 /= TYPE_STRUCTURE(type1)) then
  result := type1;
elseif (is_rec_type(type2) ? false and
  not (is_rec_type(type1) ? false)) and
  (type1 subset TYPE_STRUCTURE(type2) and
      type1 /= TYPE_STRUCTURE(type2)) then
  result := type2;
else

  $ Otherwise compress the types

  $ Scalars are simply unioned together

  result := {t1 in type1 | is_scalar_type(t1)} +
            {t2 in type2 | is_scalar_type(t2)};

  $ Sets have their component type disjuncted

  if exists etyp in type1 | gross_type(etyp) = T_SET then
    t1 := component_type(etyp);
  else
    t1 := TYPE_ERROR;                          $ No set type in type1
  end if;

  if exists etyp in type2 | gross_type(etyp) = T_SET then
    t2 := component_type(etyp);
  else
     t2 := TYPE_ERROR;                        $ No set type in type2
  end if;

  $ Only disjunct if at least one of the types had a set type

  if t1 /= TYPE_ERROR or
     t2 /= TYPE_ERROR then
     result with:= [T_SET, t1 .DIS t2];
  end if;

  $ If either type has a tuple type, merge all tuple and sequence types into a
  $ single tuple type whose component type is the disjunction of all
  $ component types.

  if exists etyp1 in type1 | gross_type(etyp1) = T_TUPLE then
     t1 := component_type(etyp1);
     (for etyp2 in type2 | gross_type(etyp2) in {T_TUPLE, T_SEQ})
        if gross_type(etyp2) = T_TUPLE then
             t1 := t1 .DIS component_type(etyp2);
        else
             t1 := t1 .DIS/component_type(etyp2);
        end if;
     end for;
     result with:= [T_TUPLE, t1];
  elseif exists etyp2 in type2 | gross_type(etyp2) = T_TUPLE then
     t2 := component_type(etyp2);
     (for etyp1 in type1 | gross_type(etyp1) in {T_TUPLE, T_SEQ})
          if gross_type(etyp1) = T_TUPLE then
             t2 := t2 .DIS component_type(etyp1);
          else
             t2 := t2 .DIS/ component_type(etyp1);
          end if;
     end for;
     result with:= [T_TUPLE, t2];

  $ Dump in all known length sequences

  else
      result +:= {etyp1 in type1 | gross_type(etyp1) = T_SEQ} +
                 {etyp2 in type2 | gross_type(etyp2) = t_SEQ};
  end if;
end if;

return result;

end op .DIS;


op .INTER(typ, valid_gross_types);

$ .INTER returns the set of elementary types in typ whose gross types
$ match any contained in valid types
```

```
result := {t in typ | gross_type(t) in valid_gross_types};
if typ = TYPE_GENERAL then
   result := TYPE_GENERAL;
end if;

return result;

end op .INTER;


proc pretty_print (typ);

pprint (typ, 3);

end proc pretty_print;


proc pprint (typ, indent);

prefix := '   ';

(for etyp in typ)
   if is_scalar_type(etyp) then
      print (' '*indent, prefix, etyp);
   else
      print (' '*indent, prefix, gross_type(etyp));
      if gross_type(etyp) = T_SEQ then
         comptyp := component_type(etyp);
         (for ctyp = comptyp(i))
             pprint (ctyp, indent+5);
         end for ctyp;
      else
         pprint (component_type(etyp), indent+5);
      end if;
   end if;

   prefix := ' | ';

end for etyp;

end proc pprint;

end program type_finder;
```

CHAPTER 5

Examples

5. Examples

5.1. Example 1: Building a linked list using tupleformers

The following example constructs a LISP-like list of the form:

[5, [5, ...]]]

Note the alternand T_OM in the recursive type structure. This is due to the typefinder not employing the type information present in the test 'if l = om'.

This example as well as all the others are the output of the implementation presented in the preceding chapter. The SETL code was hand-transcribed into quadruples intermediate code, suitable for optimization purposes. This together with the original SETL source (treated as a comment) and UD-chain information are used as input. The Q1 source code is deleted from the output to make it clearer. Dummy conditions (i.e. while 6 = 6) are inserted for the while loop for simplicity of the transcription for this and most of the following examples.

The format of the output is as follows. The source code is listed together with the set of recursive types discovered by the typefinder and the types of the variables in the program. As an example, the type:

```
  T_OM
| T_INTEGER
| T_SEQ
    T_INTEGER
    REC_#1
```

denotes either **om**, or an **integer**, or a fixed-length tuple whose first component is an **integer** and whose second component is of type **REC_#1** (a recursive type symbol).

Original source code:

```
program example1;
  l := om;
  (while 6 = 6)
    if l = om then
       l := 5;
    else
       l := [5, l];
    end if;
  end while;
end program example1;
```

Recursive Type Structures

```
{ REC_#1 } =
        T_OM
     | T_SEQ
           T_INTEGER
           REC_#1
     | T_INTEGER
```

Types

```
6  :
        T_INTEGER
OM  :
        T_OM
t1  :
        T_BOOLEAN
t2  :
        T_BOOLEAN
```

```
l  :
        T_SEQ
           T_INTEGER
           REC_#1
     | REC_#1
     | T_OM
     | T_INTEGER
```

5.2. Example 2: A pair of mutually recursive structures

This example shows how the typefinder deals with mutually recursive structures. Note how only one of the recursive types (REC_#1) has an escape clause; REC_#2 is defined solely in terms of REC_#1.

Original source code:

```
program example2;
 t := 3;
 (while 3 = 3)
    s := ['A', t];
    t := [3, s];
 end while;
end program example2;
```

Recursive Type Structures

```
{ REC_#1 }  =
        T_INTEGER
     | T_SEQ
             T_INTEGER
             REC_#2
{ REC_#2 }  =
        T_SEQ
             T_STRING
             REC_#1
```

Types

```
3  :                                t  :
        T_INTEGER                           REC_#1
s  :                                t1  :
        T_SEQ                               T_BOOLEAN
             T_STRING               "A"  :
             REC_#1                         T_STRING
     | REC_#2
```

5.3. Example 3: The with operator: recursion on the first operand

This and the next two examples deal with the **with** operator. In #3, the recursion is on the first operand of the **with** operation, in #4 the recursion is on the second, while in #5, the recursion is on both. The first operand, when recursive, generates a dynamic tuple, while the second operand generates a recursive structure. The typefinder distinguishes between these two instances, producing the appropriate types in each case.

Original source code:

```
program example3;
r := [];
(while 6 = 6)
  r := r with 3;
end;
end program example3;
```

Recursive Type Structures

```
{ REC_#1 } =
      T_NULLTUP
    | T_TUPLE
          T_INTEGER
```

Types

```
6 :
      T_INTEGER
3 :
      T_INTEGER
r :
      REC_#1
    | T_TUPLE
          T_INTEGER
    | T_NULLTUP
```

```
NULLTUP :
      T_NULLTUP
t1 :
      T_BOOLEAN
```

5.4. Example 4: The with operator: recursion on the second operand

Original source code:

```
program example4;
r := [];
(while 6 = 6)
  r := [3] with r;
end;
end program example4;
```

Recursive Type Structures

```
{ REC_#1 } =
      T_NULLTUP
    | T_SEQ
          T_INTEGER
          REC_#1
```

Types

```
6  :
      T_INTEGER
3  :
      T_INTEGER
r  :
      REC_#1
    | T_SEQ
          T_INTEGER
          REC_#1
    | T_NULLTUP
```

```
NULLTUP  :
      T_NULLTUP
t1  :
      T_BOOLEAN
t2  :
      T_SEQ
          T_INTEGER
```

5.5. Example 5: The with operator: recursion on both operands

Original source code:

```
program example5;
r := [];
(while 6 = 6)
  r := r with r;
end;
end program example5;
```

Recursive Type Structures

```
{ REC_#1 } =
      T_NULLTUP
    | T_TUPLE
          REC_#1
{ REC_#2 } =
      REC_#1
```

Types

```
6 :
      T_INTEGER
r :
      REC_#1
    | T_TUPLE
          REC_#1
    | T_NULLTUP
```

```
NULLTUP :
      T_NULLTUP
t1 :
      T_BOOLEAN
```

5.6. Example 6: A structure constructed via component assignment

In this example, a recursive structure is built by modifying only the first component in a tuple.

Original source code:

```
program example6;
 l := [5, 5];
 (while 5 = 6)
  l(1) := l;
 end;
end program example1;
```

Recursive Type Structures

```
{ REC_#1 } =
      T_SEQ
          T_INTEGER
        | REC_#1
          T_INTEGER
    | T_SEQ
          T_INTEGER
          T_INTEGER
{ REC_#2 } =
      REC_#1
```

Types

```
1  :
      T_INTEGER
6  :
      T_INTEGER
5  :
      T_INTEGER
t1  :
      T_BOOLEAN
```

```
l :
      REC_#1
    | T_SEQ
          T_INTEGER
          T_INTEGER
    | T_SEQ
          T_INTEGER
        | REC_#1
          T_INTEGER
```

5.7. Example 7: A binary tree - uniform node structure

The next two examples construct binary trees. In #6, all nodes are of uniform format, and thus leaf nodes are themselves tuples. A typical tree of such format is:

[1 [2 [3]] [4]] In #7, the leaves are kept as simple integers at the same tuple level as their parents. I.e.:

[1 [2 3] 4]. The typefinder again distinguishes between these two formats.

In #6, the data element of each node (i.e. the first component) is found by the initialization pass to be self-dependent and as such it is assigned a recursive type symbol. This symbol can be easily removed by a postprocessing pass in which recursive symbols whose structures contain no recursive symbols can be directly replaced by their structures.

Original source code:

```
program example7;
t := om;
(while 5 = 6)
  t := insert(t, 3);
end while;

proc insert(tree, info);
  if tree = om then
    return [info];
  else
    [value, left, right] := tree;
    if value < info then
      left := insert(left, info);
    else
      right := insert(right, info);
    end if;
    return [value, left, right];
  end if;
end proc insert;
end program example7;
```

Recursive Type Structures

```
{ REC_#3 }  =
        T_SEQ
            REC_#1
            REC_#3
            REC_#3
      | T_OM
      | T_SEQ
            T_INTEGER
{ REC_#2 }  =
        REC_#3
{ REC_#1 }  =
        T_INTEGER
```

Types

```
3  :
       T_INTEGER
info  :
       T_INTEGER
```

```
tree  :
       T_SEQ
           REC_#1
           REC_#3
           REC_#3
       | T_OM
       | REC_#3
       | T_SEQ
           T_INTEGER
OM  :
       T_OM
1  :
       T_INTEGER
2  :
       T_INTEGER
5  :
       T_INTEGER
6  :
       T_INTEGER
value  :
       REC_#1
     | T_INTEGER
```

```
t1  :
       T_BOOLEAN
t2  :
       T_BOOLEAN
left  :
       REC_#3
t  :
       T_OM
     | T_SEQ
           REC_#1
           REC_#3
           REC_#3
     | T_SEQ
           T_INTEGER
right  :
       REC_#3
t3  :
       T_BOOLEAN
funcval  :
       T_SEQ
           T_INTEGER
     | T_SEQ
           REC_#1
           REC_#3
           REC_#3
```

5.8. Example 8: A binary tree: nonuniform node structure

See example #6 for explanatory remarks. Note that the data elements are classified as a recursive structure whose format is identical to that of the other two node elements. This is due to not exploiting the typetesting information present in the program. This problem is discussed in Section 2.10.

Original source code:

```
program example8;
  t := om;
  s := {1 ,2, 5, 7}
  (while s /= {})
    x := arb s;
    s := s less x;
    t := insert(t,x);
  end while;


proc insert(tree, info);

  if tree := om then
    return info;
  elseif is_integer tree then
    if info < tree then
        return [tree, info];
    else
        return [tree, om, info];
    end if;
  else
    [value, left, right] := tree;
    if info < value then
        left := insert(left, info);
    else
        right := insert(right, info);
    end if;
    return [value, left, right];
  end if;

end procedure insert;
end program example8;
```

Recursive Type Structures

```
{ REC_#5 } =
      T_OM
    | T_SEQ
          REC_#5
          REC_#5
          REC_#5
    | T_SEQ
          REC_#5
          T_INTEGER
    | T_INTEGER
    | T_SEQ
          REC_#5
          T_OM
          T_INTEGER
{ REC_#4 } =
      REC_#5
{ REC_#1 } =
      REC_#5
{ REC_#2 } =
      REC_#5
{ REC_#3 } =
      REC_#5
```

Types

```
info :
      T_INTEGER
```

```
tree :
      T_OM
```

```
    | REC_#5
    | T_SEQ
          REC_#5
          REC_#5
          REC_#5
    | T_INTEGER
    | T_SEQ
          REC_#5
          T_INTEGER
    | T_SEQ
          REC_#5
          T_OM
          T_INTEGER
1  :
      T_INTEGER
OM  :
      T_OM
2  :
      T_INTEGER
3  :
      T_INTEGER
value  :
      REC_#5
5  :
      T_INTEGER
7  :
      T_INTEGER
left  :
      REC_#5
s  :
      T_SET
          T_INTEGER
    | T_NULLSET
t  :
      T_OM
    | T_SEQ
          REC_#5
          REC_#5
          REC_#5
    | T_INTEGER
    | T_SEQ
          REC_#5
          T_INTEGER
    | T_SEQ
          REC_#5
          T_OM
          T_INTEGER
t0001  :
      T_BOOLEAN
NULLSET  :
      T_NULLSET
t0003  :
      T_BOOLEAN
t0002  :
      T_BOOLEAN
right  :
      REC_#5
t0004  :
      T_BOOLEAN
t0005  :
      T_BOOLEAN
x  :
      T_INTEGER
funval  :
      T_SEQ
          REC_#5
          REC_#5
          REC_#5
    | T_SEQ
          REC_#5
          T_INTEGER
    | T_INTEGER
    | T_SEQ
          REC_#5
          T_OM
          T_INTEGER
```

5.9. Example 9: a simple recursive descent parser

Example 8 is a typefind of a more substantial program. The language is a trivial one, expressions and left hand sides of assignments restricted to single tokens. In order to bypass the complexity of incorporating into the typefinder information regarding built-in functions, the break function used in the token generation routine was simulated with the assignment of a string constant to the return value. A working optimizer would of course have a table of built-in functions together with their predefined return values. As with all the other examples, since the intermediate code was hand-transcribed, unique names were assigned to the various local variables to replace name-resolution, which is normally performed by a front-end.

Original source code:

```
program simple_parser;

$ General tree construction program.
$
$ Program performs a recursive descent parse of a simple language,
$  building the parse tree as it goes.
$
$ Language syntax:
$
$ <program> ::=
$     <dcl statement>
$     <exec or label statement>
$     END.
$
$ <dcl statement> ::=
$     DECLARE <identifier list>
$
$ <exec or label statement ::=
$     <goto statement>
$   | <if statement>
$   | <assignment statement>
$
$ <identifier list> ::=
$     identifier
$   | identifier , <identifier list>
$

var
  token;
```

```
read (source);
parse_program;


proc parse_program;

$ Top-level recursive descent routine

token := get_token();

$ Initialize parse tree

tree := [];

$ Parse to eof

(while token /= om)
  tree with:= parse_statement();
end;

$ Build top-level node

parse_tree := ['PROGRAM', tree];

end proc parse_program;


proc parse_statement;

$ Parse source statement

case token of
  ('DECLARE'):
    token := get_token();
    result := parse_declare();

  ('IF'):
    token := get_token();
    result := parse_if();

  ('GOTO'):
    token := get_token();
    result := parse_goto();

  else

    $ No keyword - assignment

    result := parse_assignment();

end case token of;

return result;
```

```
end proc parse_statement;


proc parse_declare;

token := get_token();

$ Build result node

result := ['DECLARE', parse_identifier_list()];

return result;

end proc parse_declare;


proc parse_identifier_list;

$ Init result to 1st id in list

result := [token];

$ Add any remaining id's

(while (token := get_token()) = ',')
   token := get_token();
   result with:= token;
end while;

return result;

end proc parse_identifier_list;


proc parse_goto;

$ Next token is label

result := ['GOTO', [get_token()]];

$ Get back into token synch for statement parser

token := get_token();

return result;

end proc parse_goto;


proc parse_assignment;

lhs := token;
```

```
$ Next token is :=

token := get_token();
rhs := get_token();
result := [':=', [lhs, rhs]];

return result;

end proc parse_assignment;


proc parse_if;

$ Left hand side of cond

exp1 := get_token();

$ Relational operator

rel  := get_token();

$ Right hand side of cond

exp2 := get_token();

$ Build result node - Note recursive to statement parser

result := ['IF', [exp1, rel, exp2, parse_statement()]];

return result;

end proc parse_if;


proc get_token;

token := break(source, ' ');

return token;

end proc get_token;

end program simple_parser;
```

Recursive Type Structures

```
{ REC_#3 }  =                               T_STRING
      T_SEQ                                 T_SEQ
          T_STRING                              T_STRING
          T_SEQ                                 T_STRING
              T_STRING                    | T_SEQ
    | T_SEQ                                 T_STRING
```

```
      T_TUPLE
         T_STRING
  | T_SEQ
      T_STRING
      REC_#4
{ REC_#2 } =
   T_TUPLE
      T_STRING
```

```
{ REC_#1 } =
    T_NULLTUP
  | T_TUPLE
       T_SEQ
          T_STRING
          T_SEQ
             T_STRING
     | T_SEQ
          T_STRING
          T_SEQ
             T_STRING
             T_STRING
     | T_SEQ
          T_STRING
          REC_#4
     | T_SEQ
          T_STRING
          T_TUPLE
             T_STRING
{ REC_#4 } =
    T_SEQ
       T_STRING
       T_STRING
       T_STRING
       REC_#3
```

Types

```
MAIN_source :
    T_GENERAL
OM :
    T_OM
PARSE_IDENTIFIER_LIST_result :
    T_TUPLE
       T_STRING
  | REC_#2
MAIN_parse_tree :
    T_SEQ
       T_STRING
       T_NULLTUP
     | T_TUPLE
          T_SEQ
             T_STRING
             T_SEQ
                T_STRING
        | T_SEQ
             T_STRING
             T_SEQ
                T_STRING
                T_STRING
        | T_SEQ
             T_STRING
```

```
                REC_#4
        | T_SEQ
             T_STRING
             T_TUPLE
                T_STRING
GET_TOKEN_funval :
    T_STRING
MAIN_token :
    T_STRING
NULLTUP :
    T_NULLTUP
MAIN_t0001 :
    T_BOOLEAN
PARSE_IF_rel :
    T_STRING
PARSE_IF_funval :
    T_SEQ
       T_STRING
       REC_#4
PARSE_IF_exp1 :
    T_STRING
PARSE_IF_exp2 :
    T_STRING
PARSE_GOTO_funval :
        T_SEQ
            T_STRING
            T_SEQ
                T_STRING
PARSE_IF_temp01  :
        T_SEQ
            T_STRING
            T_STRING
            T_STRING
            REC_#3
      | REC_#4
","  :
        T_STRING
" "  :
        T_STRING
":="  :
        T_STRING
"IF"  :
        T_STRING
PARSE_GOTO_temp01  :
        T_SEQ
            T_STRING
PARSE_DECLARE_funval  :
        T_SEQ
            T_STRING
            T_TUPLE
                T_STRING
"GOTO"  :
        T_STRING
PARSE_STATEMENT_funval  :
        T_SEQ
            T_STRING
            T_SEQ
                T_STRING
                T_STRING
      | T_SEQ
            T_STRING
            T_SEQ
                T_STRING
      | REC_#3
      | T_SEQ
            T_STRING
            REC_#4
      | T_SEQ
            T_STRING
            T_TUPLE
                T_STRING
PARSE_ASSIGNMENT_funval  :
        T_SEQ
            T_STRING
            T_SEQ
                T_STRING
                T_STRING
PARSE_IF_result  :
        T_SEQ
            T_STRING
            REC_#4
"PROGRAM"  :
        T_STRING
"DECLARE"  :
        T_STRING
PARSE_GOTO_result  :
        T_SEQ
            T_STRING
            T_SEQ
                T_STRING
PARSE_ASSIGNMENT_temp01  :
        T_SEQ
            T_STRING
            T_STRING
PARSE_ASSIGNMENT_rhs  :
        T_STRING
PARSE_ASSIGNMENT_lhs  :
        T_STRING
PARSE_IDENTIFIER_LIST_funval  :
        T_TUPLE
            T_STRING
PARSE_DECLARE_result  :
        T_SEQ
            T_STRING
            T_TUPLE
                T_STRING
MAIN_tree  :
        T_TUPLE
            T_SEQ
                T_STRING
                T_SEQ
                    T_STRING
          | T_SEQ
                T_STRING
                T_SEQ
                    T_STRING
                    T_STRING
          | T_SEQ
                T_STRING
                REC_#4
          | T_SEQ
                T_STRING
                T_TUPLE
                    T_STRING
      | REC_#1
      | T_NULLTUP
PARSE_STATEMENT_result  :
        T_SEQ
            T_STRING
            T_SEQ
```

```
                T_STRING
                T_STRING
      | T_SEQ
            T_STRING
            T_TUPLE
                T_STRING
      | T_SEQ
            T_STRING
            REC_#4
      | T_SEQ
            T_STRING
            T_SEQ
                T_STRING
PARSE_ASSIGNMENT_result  :
        T_SEQ
            T_STRING
            T_SEQ
                T_STRING
                T_STRING
PARSE_IDENTIFIER_LIST_t0001  :
        T_BOOLEAN
```

CHAPTER 6

Conclusions and Further Research

6. Conclusions and Further Research

The proposals presented in this thesis are part of an attempt to increase the usefulness of SETL. One of the primary problems in programming in SETL is the overhead imposed by the flexibility afforded the programmer in terms of weak typing. We wish to allow the programmer this flexibility of programming in a very high-level language and have the language processor implement the algorithm in as efficient a manner as possible. The ideal level of efficiency would be that of a corresponding program written in C. Typefinding is an important part of this high to low-level transformation as it allows us to bind variables to their types at translation time rather than during execution.

When presented with a program containing no recursive data structures, Tenenbaum's typefinding algorithm performs satisfactorily, correctly determining exact types (i.e. the same types the programmer would himself supply) for the majority of variables. The usefulness of the analysis is readily apparent: the removal of a large portion of run-time type checks from a program. However, for programs containing one or more recursive structures, the algorithm is unable to analyze the type structure of the program in any reasonable fashion. This is because the recursive structure is typically a central object of the program from which many variables receive their values, directly or indirectly. As the algorithm is unable to assign any sort of a precise type to the structure, all such dependent variables also remain either untyped or are given some type that is overly conservative (e.g. **tuple(general)** as opposed to **tuple(integer)**).

The typefinder presented in this thesis is capable of uncovering and precisely typing such recursive structures, assigning them the same recursive types that the programmer might. This capability greatly enlarges the class of SETL programs which can be reasonably typed, as recursive programming using nested tuples is quite natural in SETL. In addition to assigning the recursive structure a precise type, variables that extract values from the structure can be strictly typed (i.e. given a type with a single alternand) in many cases allowing for more efficient code to be generated for operations on these variables.

Another consequence of the algorithm is the near disappearance of type **general** from the resulting type equations. With the exception of input variables and parameter and result types of external routines which are not explicitly typed by the programmer, **general** need not be introduced into the equations. Of course there may be situations when a type is sufficiently unorthodox (e.g. **integer** | **tuple(real)** | **set(string)**) that we may decide to replace it by **general**, but this is our choice and is not a prerequisite to guarantee termination of the analysis as in Tenenbaum's typefinder. In addition, statically complex structures are no longer mistaken for nonconvergent but rather can be given their precise type.

It still remains to be determined whether the approach used in the algorithm can be applied to the backwards pass as well, allowing for the inference of recursive structures from usage as well as definition. As many large systems written in SETL consist of successive phases which communicate via intermediate structures written out to secondary storage, it is desirable to be able to read in an already constructed recursive structure from and determine its structure from how it is used.

A large number of imprecise types assigned to variables by the typefinder (both Tenenbaum's as well as the one presented here) result from the typefinder not employing information available in conditional statements that test the type of a variable. If such information can be obtained, and assuming the backwards pass can be subjected to an analysis similar to the one we have presented for the forward pass, and the types of input variables and external routines are specified by the programmer, then imprecise typing should result only from type assignments dependent upon the semantics of the program. For example in the following fragment:

```
if somecondition then
   x := 0;
   y := 4;
else
   x := 1;
   y := {4};
end if;
   ....              $ No redefinitions of x or y here
if x = 0 then
   z := y + 3;
else
   z := y + {3};
end if;
```

though the subsequent uses of y are correct and require no run-time check, it requires a highly sophisticated optimizer to detect this fact.

In addition to the ability to strictly type a larger number of variables, we present several approaches towards exploiting the typing of recursive structures with respect to efficient storage management. We plan to research further into this area as part of an overall effort to develop a lower-level, more efficient language processor for the SETL language.

A analysis closely related to typefinding is that of range analysis. Indeed, strongly typed languages (Ada for example) often consider the size and dimensions of composite objects to be part of their type. If the notion of type in SETL can be extended to include the size of an object, and we can analyze the size of composite objects such as sets, tuples and maps, we can allocate the space necessary for such an object at one time (possibly upon initial program entry) and remove the overhead of checking for available space at the point of assignment to the object.

The second portion of this thesis proposes several extensions to the current data representation sublanguage. Many of them arise from the fact that recursive structures can now be typed. The primary goal in presenting these extensions is to allow the programmer a more expressive environment within which he can precisely type the identifiers of his program.

Experience with representation declarations in SETL shows us that to obtain the maximum effect of the representation sublanguage, the programmer must design his program keeping in mind the eventual goal of adding declarations. If not he will find himself redesigning the eventual algorithm to allow for a satisfactory representation. It remains to be seen whether a truly transparent data representation sublanguage is feasible, one which allows the programmer to design an algorithm with no prior thought given to the declarations and representations eventually to be added.

A final, and perhaps the most controversial question is the future direction SETL should take with regard to its type model. We plan to reexamine the type model of SETL in light of the recent concern with program reliability and strong typing. The primary appeal of SETL as a programming language is its use of the set and mappings as primitive structures and its lack of required declarations. These two features enable the programmer to concentrate upon the solution to the abstract algorithm and enables SETL to be used as a powerful prototyping language. Though the weak typing of the language does occasionally prove useful (e.g. recursive structures and arbitrary length heterogeneous arrays such as the multi-type stack used in parsing algorithms), it is often employed in a sloppy ad-hoc fashion, resulting in an eventual redesigning of the algorithm to gain efficiency or readability.

In addition, as is readily seen by the algorithm presented in this paper, though fairly exact typefinding can be performed upon a weakly typed language, the overhead in doing so is prohibitive. Compared with the type unification scheme used in both B and ML, our algorithm is both expensive and unduly complex. It therefore seems to follow from our result that to operate efficiently in an environment in which typechecking aids the programmer, the programming language should be strongly typed. The type model of B, typing by initial assignment, is an attractive alternative, freeing the programmer from the burden of declarations while simultaneously preventing him from being careless with the type structure of his program. However, recursive structures whose shapes vary dynamically are prohibited in B due to its one-type-per-variable restriction, as are all other type unions, some of which prove quite useful (any form of variant structure for example). A more satisfactory type model, one which allows both unions as well as recursive structures, while at the same time keeping the number of necessary declarations to a minimum, is the type model of ML. We thus feel that any subsequent redesign of the SETL type model should use the type model of ML as a foundation.

Bibliography

Ada83
Military Standard Ada Programming Language ANSI/MIL-STD-1815A January 1983

Card85
Cardelli, L. *Basic Polymorphic Typechecking* Polymorphism - The ML/LCF/HOPE Newsletter 2,1 (Jan. 85)

Dewa79
Dewar, R. B. K., A. Grand, C. S. Liu, E. Schonberg, J. T. Schwartz *Programming by refinement, as exemplified by the SETL representation sublanguage.* ACM Trans. Program. Lang. Syst. 1,1 (July 1979) pp. 27-49

Dewa81
Dewar, R. B. K., E. Schonberg, J. T. Schwartz *Higher Level Programming: An Introduction to the Use of the Set- Theoretic Programming Language SETL.* Courant Inst. of Mathematical Sciences, New York Univ., New York, N.Y. 1981

Fong75
Fong, A. C., J. B. Kam, J. D. Ullman *Applications of lattice algebra to loop optimization.* In Conf. Rec. 2nd ACM Symp. on Principles of Prog. Lang., Palo Alto, Calif., January 20-22, 1975, pp 1-9

Freu83
Freudenberger, S., J. T. Schwartz, M. Sharir *Experience with the SETL optimizer.* ACM Trans. Program. Lang. Syst. 5,1 (January 1983), pp 26-45

Freu84
Freudenberger, S. *On the Use of Global Optimization Algorithms for the Detection of Semantic Programming Errors.* Courant Computer Science Rep. 24, Courant Inst. of Mathematical Sciences, New York Univ., New York, N.Y. 1984

Hech77
Hecht, M. S. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, N.Y. 1977

Hoar75
Hoare, C. A. R. *Recursive Data Structures* International Journal of Computer and Information Sciences 4,2 (1975)

Jone76
Jones, N. D., S. S. Muchnick *Binding time optimization in programming languages: some thoughts toward the design of an ideal language.* In Conf Rec. 3rd ACM Symp. on Principles of Program. Lang., Atlanta, Ga., January 19- 21, 1976 pp 77-94

Jone81
Jones, N. D., S. S. Muchnick *Flow Analysis and Optimization of LISP-like structures.* in

Program Flow Analysis: Theory and Applications, S. S. Muchnick and N. D. Jones (Editors). Prentice-Hall, Englewood Cliffs, N.J. (1981)

Kapl78
Kaplan, M. A., J. D. Ullman *A general scheme for the automatic inference of variable types.* In Conf Rec. 5th ACM Symp. on Principles of Program. Lang., Tucson, Ariz., January 23-25, 1978, pp 60-75

Kost69
Koster, C.H.A. *On Infinite Modes* Algol Bulletin 30 in SIGPLAN Notices 4,3 1969

Kruc84
Kruchten, P., E. Schonberg, J. T. Schwartz *Software Prototyping Using the SETL Programming Language.* IEEE Software 1,4 (October 1984) pp 66-75

McCa65
McCarthy, J., S. Levin *LISP 1.5 Programmer's Manual* 2nd edition M.I.T. Press, Cambridge , Mass., 1965

Meer83
Meertens, L. *Incremental Polymorphic Typechecking in B.* Conf. Record 10th ACM Symposium Principles of Prog. Languages (1983)

Meer85
Meertens, L., S. Pemberton *Description of B.* SIGPLAN Notices 20,2 (1985) pp 58-76.

Miln83
Milner, R. *A Proposal for Standard ML.* Polymorphism - The ML/LCF/HOPE Newsletter 1,3 (Dec. 1983)

PL/I76
OS PL/I Checkout and Optimizing Compilers: Language Reference Manual. IBM GC33-0009-4, 5th ed., 1976

Scho79
Schonberg, E., J. T. Schwartz, M. Sharir *Automatic data structure selection in SETL.* In Conf. Rec. 6th ACM Symp. on Principles of Program. Lang., San Antonio, Texas, January 29-31, 1979, pp 197-210.

Schw75a
Schwartz, J. T. *Optimization of very high level languages, part I: Value transmission and its corollaries.* Computer Lang. 1,2 (June 1975), pp. 161-194

Schw75b
Schwartz, J. T., *Use-use chaining as a technique in typefinding.* SETL Newsletter 140, Courant Inst. of Mathematical Sciences, New York Univ., New York, N.Y. 1975

Shar78

Sharir, M., *A few cautionary notes on the convergence of iterative data-flow analysis algorithms*. SETL Newsletter 208, Courant Inst. of Mathematical Sciences, New York Univ., New York, N.Y. 1978

Tene74

Tenenbaum, A. M. *Type determination for very high level languages*. Courant Computer Science Rep. 3, Courant Inst. of Mathematical Sciences, New York Univ., New York, N.Y. 1974

Wirt71

Wirth, N. *The Programming Language Pascal*. Acta Informatica 1,1 (1971) 35-63

NYU COMPSCI TR-201
Weiss, Gerald
Recursive data types in
SETL c.1

NYU COMPSCI TR-201
Weiss, Gerald
Recursive data types in
SETL c.1

	[illegible]
NOV 0 4 1986	[illegible]
NOV 1 9 1986	RENEWAL