Computer Science Department

TECHNICAL REPORT

"Typefinding Recursive Structures: A Data-Flow Analysis in the Presence of Infinite Type Sets"†

by

Gerald Weiss

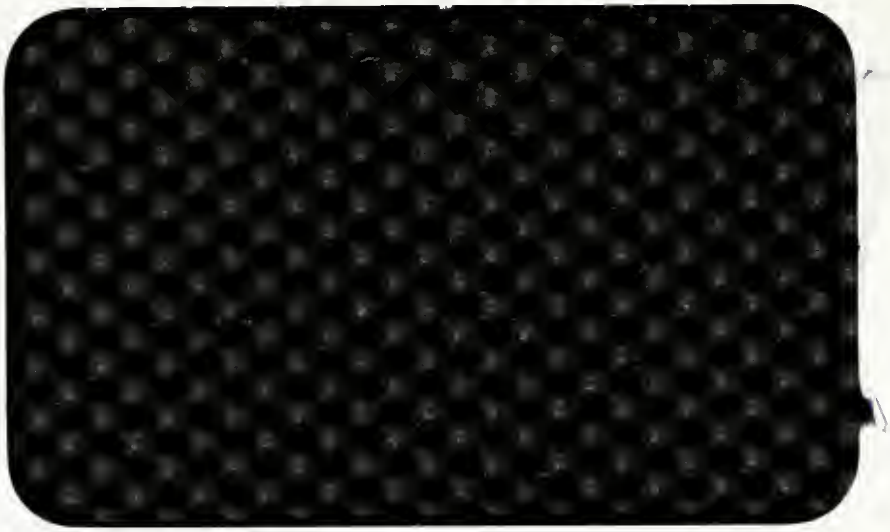
Edmond Schonberg

Technical Report #235

August, 1986

NEW YORK UNIVERSITY

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



"Typefinding Recursive Structures: A Data-Flow Analysis in the Presence of Infinite Type Sets"†

by

Gerald Weiss
Edmond Schonberg

Technical Report #235
August, 1986

---

† This work was supported in part by the Office of Naval Research Grant NO0014856K0413.

TYPEFINDING RECURSIVE STRUCTURES: A DATA-FLOW ANALYSIS IN THE PRESENCE OF INFINITE TYPE SETS.†,††

Gerald Weiss* - Edmond Schonberg**

* Department of Computer and Information Science, Brooklyn College / CUNY
** Department of Computer Science, Courant Institute of Mathematical Sciences

ABSTRACT

Very-high-level languages which are declaration free often incorporate some form of type inference system to allow for compile-time type determination. In order to guarantee algorithmic termination, typefinding systems employing data-flow analysis have been unable to deduce the types of recursive structures (such as trees). As these structures are often central to an algorithm, a substantial amount of type information is lost as a result. We present a method of performing type analysis upon such objects which detects the presence of recursive structures in a program and determines the internal structure of such objects. Furthermore, the method is compatible with standard data-flow analysis techniques.

1. Introduction

Very high level languages are often weakly typed: different occurrences of a name can be associated with values of 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. One class of such algorithms employs data-flow analysis to perform type inference. Due to problems of algorithmic termination, however, these algorithms have been unable to perform well in the presence of variables that can assume an arbitrary number of types, in particular recursive structures such as trees. In this paper we present a typefinding algorithm that both detects the presence and uncovers the structure of such objects. Furthermore, the method presented is compatible with the standard data flow analysis framework.†††

We present our algorithm for the programming language SETL. SETL[2] 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.

Section 2 discusses previous work in this area. Section 3 introduces the basic type model of SETL and programming with recursive data types in that language. Section 4 presents an informal introduction to typefinding as performed in SETL. Section 5 presents our method, compares it to the existing typefinder and discusses applicable optimizations, and gives a brief overview of an implementation of the algorithm, and section 6 summarizes our results. An appendix containing some example type analyses follows the paper.

2. Previous Work

2.1. Data-Flow Methods

Initial work on typefinding was done by Tenenbaum[10] and his algorithm is the basis of the typefinder used in the current SETL optimizer. Jones and Muchnick[3], in presenting the design philosophies of a programming language with late binding times, also develop a typefinding algorithm that is simpler and more general than Tenenbaum's, but requires more storage, as the resulting system of equations is larger. Kaplan and Ullman[5] also develop a general method of typefinding and show that their algorithm is yet more

---

[†] To appear in the proceedings of the 1986 IEEE International Conference on Computer Languages.

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

[†††]A more comprehensive treatment of the subject together with a discussion of the supplementary declarative sublanguage of SETL can be found in the first author's doctoral thesis[11].

comprehensive than Tenebaum's or Jones and Muchnick's

As the above algorithms employ data-flow and iterate over the program until convergence, they are unable to correctly type recursive structures such as trees or linked lists, because the set of possible types assumable by such structures is infinite and therefore the algorithms as given do not terminate Jones and Muchnick[4] subsequently developed a method to discover the type structure of various LISP-like objects They accomplish this using tree grammars which handle the infinite systems as well as the finite ones The method is both expensive and requires a framework other than that of standard data-flow analysis.

2.2. Type Unification

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

ML[8], is a strongly typed language in which most declarations are optional A typefinding method for ML[1] has been developed based upon Robinson's unification algorithm The method performs tree equivalencing of types, checking types for consistency without employing program flow information Meertens[6] has developed independently a similar method for the programming language B[7] With regard to recursive data structures, ML requires explicit declarations of recursive data types, while B disallows structures whose depths vary dynamically

3. 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[7], for example, which is strongly typed is nevertheless declaration-free

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 therefore can not be typechecked Furthermore, the set of possible shapes such structures can assume may be infinite requiring the introduction of declaration machinery There is no problem if pointers exist in the language but SETL has no such notion

3.1. 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

3.2. Programming with Recursive Structures in SETL

SETL allows for both implementations of recursive structures: pointer and value oriented The high-level form of recursive structure representation is obtained 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 attached to 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

In addition, pointer-oriented recursive structures can be implemented in SETL While essentially a value-semantics language, the flavor of pointers can be gotten through use of atoms and mappings (both of which are SETL primitives) 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

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 available as language primitives 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),

$ Empty Tree
if tree = om then
  z := newat,
  INFO(z) := x,
  tree := z,
  return,
$Left Insertion
elseif x < INFO(tree) then
  $ Leaf Node
  if LEFT(tree) = om then
    z := newat,
    INFO(z) := x,
    LEFT(tree) := z,
    return,
  $ Must descend further
```

```
    else
       insert(LEFT(tree), x),
    end if LEFT(tree),
  $ Right Insertion
  else

  end if x < INFO(tree),
  end proc insert,
```

Upon examination of these two modes of programming recursive data structures, it becomes obvious that the pointer-oriented method is the more efficient due to the less amount of run-time allocation required, and copies of subtrees are not constantly being created (many of which are needed only temporarily and therefore become garbage). 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 as the programmer can inadvertently assign the same pointer to several nodes. 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, problems which cannot occur when programming in a functional value-oriented style.

The above makes 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 this paper provides us with information that assists us in this transformation. Obtaining underlying structural information regarding a recursive type programmed in the high-level fashion is the first step toward an automatic transformation of the object (and its corresponding code) into a more efficient pointer-based representation.

4. Data-Flow Typefinding in SETL

4.1. Terminology

The predefined elementary or primitive data types of SETL are **integers, reals, strings, atoms** and **om** (the undefined value). Two additional types are introduced for the purposes of typefinding: **general** and **error**, the first being the type about which nothing is known, the second representing an erroneous type, i.e. uninitialized value or incompatible usages. These primitive types are combined into more complex type descriptors using **set**, **sequence** and **tuple** constructors, the latter two denoting arbitrary- and fixed-length sequences respectively.

As SETL is weakly typed, we also allow for arbitrary type unions. Given types t1 and t2, their **alternation**, denoted **t1 | t2**, is the type consisting of the union of the sets of domain values valid for t1 and t2.

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**).

4.2. An Informal Introduction to Data-Flow Typefinding in SETL

The basic technique of data-flow 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. For example:

```
(1)   a := 3,
(2)   read (b),
(3)   y := a + b,
```

The assignment of an integer to a in statement 1 is carried forward to statement 3 where it is used to determine that y must be an integer (typefinding assumes that the program is correct). Working backwards, we determine that b in statement 3 must also be an integer and thus the value read in 2 must be an integer.

It is possible for a variable in a SETL program to assume a potentially infinite number of values of distinct types. For example, the statement

$$s := \{s\},$$

assigns to s the singleton set whose element is the previous value of s. (Associated with every operation is a *propagation function* which determines the result type of the operation given the types of the inputs. For example, the result type of a set-forming operation, assuming input type **t** is **set(t)**.) If the type of s is **integer** prior to executing the statement, its type after the statement becomes **set(integer)**. Furthermore, if this statement is executed within a loop, the type of s will never stabilize during the type analysis, but will rather produce the successive types

set(integer), set(set(integer)), ...

To avoid this problem of nonconvergence, data-flow typefinding algorithms group the more complex types into a single class, resulting in a finite number of distinct classes of types in the system. Tenenbaum, for example, folds any type with four or more nesting levels of constructors into a type with three levels and transforms the innermost type into **general**. Though this approach solves the convergence problem, information is lost concerning such types, an important subclass of which are recursive structures implemented as arbitrarily nested sequences or sets. Furthermore statically complex structures that exceed the nesting level are also folded and information concerning their types is lost as well.

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

The algorithm we present is workpile-based and operates in a manner similar to other data-flow typefinding algorithms.

We initially detect the set of self-embedding variable occurrences (i.e. occurrences which incorporate as a proper component their previous value and thus have a potentially recursive structure) and flag them as potentially nonconvergent. To determine the set of such occurrences, we define a relation, BREACHES*, that builds paths of instances. Intuitively, given instances i and j, i $\in$ BREACHES* j if the value of j is in any manner dependent upon the value of i. We initially define the relation

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

where UD is the use-definition mapping. We now calculate the closure, BREACHES*, which corresponds to going backwards along a path whose links are the UD chains and ovar-ivar relationships, that is, the propagation of a value from an ivar to an ovar.

The task of determining which variable occurrences are self-embedding can now be accomplished by examining each ivar 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 mark as a **recursive type symbol**. Associated with this symbol is a type structure, described by means of type constructors and allowing self-reference. When such an 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 nonterminating production of arbitrarily deep embedded symbols. To see this, consider the following code fragment.

```
(1)  s  = 3,
(2)  (while somecondition)
(3)        s  = {s},       $ s embeds itself
(4)        x from s,  $ remove an element
                                 $ from s
(5)        x  = x + 1,  $ dangerous, works
                              $ first time only
(6)  end while,
```

Traditional typefinding types the ivariable s of statement 3 (written as $s_{3.2}$) successively as

integer, set(integer), set(set(integer)), ...

The recursive structure typefinder, upon discovering that the ivariable $s_{3.2}$ 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. We thus distinguish between the type name of a recursive occurrence, which is used for embedding (i.e. as input to a type constructor), and the type structure which is employed everywhere else. The set-forming operation of statement 3 will thus have as its result type (i.e. the type of $s_{3.1}$) **set(REC_1)** regardless of the values assigned to the structure of **REC_1**. When we propagate the type of the ovariable $s_{3.1}$ to the ivariable $s_{3.2}$ (through the top of the loop), its type structure will have **set(REC_1)** added to it. However, when the the type of the ovariable $s_{3.2}$ is calculated this second time (and for all future processings of this instruction) the result type remains **set(REC_1)**, and thus the types of both ivar and ovar converge. (The determination of the type of x is discussed below.) The above method works equally well for sequences, tuples as well as mutually recursive structures.

5.1. Ensuring Complete Type Propagation

Propagating a recursive type symbol rather than its internal 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. For example, in the above code fragment, x is extracted from the set s and then used as an operand in an arithmetic operation. The propagation function for **from** is *given an input type* **set(t)**, *the result type is* **t**. The type of s_4 *is* **set(REC_1)** (it receives its type from $s_{3.2}$), and therefore x is assigned the type **REC_1**. In the next operation, the propagation function for + first checks that the operands are compatible, i.e. that x has **integer** as one of its type alternands (since the second operand is an integer), and to determine this the internal structure of **REC_1** must be examined. If a new type is added to the structure of **REC_1**, the + operation should be reprocessed as its result type may be affected by this new type information.

In order to guarantee complete propagation of type information throughout the program we define a new map **occurrences-dependent-upon**, whose domain is the set of recursive type names, and whose range is the set of occurrences. This map contains for each recursive type symbol a list of all occurrences that need to examine the internal type structure of that symbol. 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 for reprocessing, in addition to the corresponding ovariable.

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

Input: Quadruple code of SETL program together with UD chains.

1) The workpile is initialized to the set of ovariables that have a constant right hand side. A map, **occurrences-dependent-upon**, is initialized to the empty set.

2) Whenever an embedding operation (i.e. an operation that applies a type constructor, for example, set or tuple formers) is encountered, the BREACHES* of each ivariable of that operation is calculated. If an ivariable appears in its own BREACHES*, it is marked as *self-dependent*.

3) All *self-dependent* ivariables are given new, unique type names which are marked 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 map **type-structure** is set to **error** for each element in its domain (i.e. for each recursive type).

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

    5a) If the occurrence in question is an ovariable, calculate its type using the type 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 **occurrences-dependent-upon** 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 union of all definitions reaching that occurrence. This becomes the type descriptor 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 5 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.

5.2. Proof of Termination

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 **tuple**(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 formers appearing in loops can no longer expand their result types indefinitely and extraction operations (e.g. **arb**, subscripting) merely introduce types that are 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 eventually converge.

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 define 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**

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

$$nd(sequence(t1, t2, \ldots, tn)) = max(nd(t1), nd(t2), \ldots, nd(tn)) + 1$$

$$nd(t1 \mid t2 \mid \ldots \mid tn) = max(nd(t1), nd(t2), \ldots, nd(tn))$$

this function is originally introduced in Tenenbaum[10] for the purpose of placing the limit upon a depth of type symbols and thus bounding the lattice.

We similarly define the length, *ln*, of a symbol as

$$ln(scalar\_type) = ln(set(t)) = -1$$

$$ln(NULLTUP) = 0$$

$$ln(tuple(t)) = 0$$

$$ln(seq(t1, \ldots, tn)) = n$$

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. We accomplish this using 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).

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 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

$$x_{5.2} \rightarrow x_{2.1} \rightarrow a_2 \rightarrow a_1 \rightarrow 5_1$$

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

$$x_{5.2} \rightarrow x_{5.1} \rightarrow x_{5.2}$$

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 IC, 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

5.3. Completeness of the Algorithm

Completeness, i.e. insuring that 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

$$(4) \qquad x := z + 3,$$

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 will 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

5.4. Complexity of the Algorithm

With regard to the cost of the algorithm, 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 a number of iterations over the program which may be linear in the number of variable occurrences, but in general is much smaller[9]. In this regard our algorithm does no worse than Tenebaum's, and indeed depending upon the constructor nesting limit chosen by a particular implementation of Tenenbaum's algorithm, 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.

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 tasks is quadratic with respect to the number of variable instances in the program (i.e. every operation may be self-embedding, requiring the self-dependency test to be performed upon all ivariables in the program). Examination of the internal structures of a recursive type may in the worst case (again when all ivariables are self-dependent) require examination of 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 (typefinding a simple recursive descent parser generated 4 recursive symbols).

5.5. Applicable Optimizations

Typefinding of recursive data structures allows us to determine the data types of subcomponents of an object, and the relationship among those subcomponents. This allows an increased number of variables in the program to be strictly typed, not only those that are instances of the recursive structure, but variables that receive their value from the structure as well. This in turn can result in more efficient representations (e.g. a more compact type descriptor) than possible had the structure remained untyped.

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 static constructs resembling such structures) were present.

5.6. Implementation

The algorithm described in this paper has been implemented (in SETL) for all of SETL and has been run on a number of example programs, the largest being a recursive descent parser for a simple language In all cases, the resulting recursive types correspond closely to those types that a programmer would have assigned (using pointers in a Pascal-like language)

The appendix following this paper contains several sample programs together with the output produced by our implementation

6. Summary and Conclusions

The algorithm presented in this paper is part of an ongoing attempt to increase the usability of SETL One of the main sources of inefficiency of SETL is the overhead imposed by weak typing Weak typing is nevertheless indispensable in a value-oriented language which is to support recursive structures Correct typefinding of recursive structures should allow the SETL run-time system to manipulate such structures with the same efficiency as if they had been described in a lower level language, e g C or Ada

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 paper 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.

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.

References

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

[2] 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

[3] 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

[4] 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)

[5] 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

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

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

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

[9] 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

[10] 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

[11] Weiss, G *Recursive Data Types in SETL: Automatic Determination, Data Language Description, and Efficient Implementation.* PhD Thesis, New York University New York, N Y 1985

Appendix: Some Sample Typefindings

The following is a pair of programs together with the result of performing type analysis upon them using our method The SETL code was hand-transcribed into quadruples, suitable for optimization purposes Dummy conditions (e g while 6 = 6) are inserted for the while loop for simplicity of the transcription

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_TUPLE
        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)

Example 1 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 example1,
 t = 3,
 (while 3 = 3)
   s = ['A', t];
   t = [3, s],
 end while,
end program example1,
```

Recursive Type Structures

```
      T_INTEGER
    | T_TUPLE
        T_INTEGER
        REC_#2
{ REC_#2 } =
      T_TUPL
        T_STRING
        REC_#1
```

Types

```
s
      T_TUPLE
        T_STRING
        REC_#1
    | REC_#2
t
      REC_#1
```

Example 2 The **with** operator recursion on both operands

The **with** operator applied to a tuple (as its first operand) yields an identical tuple except that the second operand is appended to it a an additional final component In the statement

t = t **with** t,

the tuple t is used both as the tuple and the additional component The use of t as the first operand (in conjunction to it being the result variable) causes the typefinder to transform t from a known-length tuple (**tuple**) into an arbitrary-length **sequence** The use of t as the second operand causes a self-embedding requiring the generation of a recursive type symbol Both situations are properly handled by the typefinder

Original source code:

```
program example2,
r = [],
(while 6 = 6)
  r = r with r,
end,
end program example2,
```

Recursive Type Structures

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

Types

```
r
      REC_#1
    | T_SEQUENCE
          REC_#1
    | T_NULLTUP
```

NYU COMPSCI TR-235
Weiss, Gerald
Typefinding recursive
structures c.2