Courant Computer Science Report #3

October 1974

Type Determination for Very High Level Languages

Aaron M. Tenenbaum

Courant Institute of Mathematical Sciences

Computer Science Department

New York University

Report No. NSO-3 prepared under Grant No. NSF-GJ-1202x3 from the National Science Foundation

COURANT COMPUTER SCIENCE PUBLICATIONS

COURANT COMPUTER SCIENCE NOTES

	Price
On Programming: An Interim Report on the SETL Project. Installment I: Generalities J. T. Schwartz, *1973, vii+160 pp.*	$ 4.25
Installment II: The SETL Language and Examples of Its Use J. T. Schwartz, *1973, viii+520 pp.*	13.00
A SETLB Primer. H. Mullish and M. Goldstein, *1973, v+201 pp.*	5.25
Combinatorial Algorithms, E. G. Whitehead, Jr., *1973, vi+104 pp.*	2.75

COURANT COMPUTER SCIENCE REPORTS

No. 1 ASL: A Proposed Variant of SETL
Henry Warren, Jr., *1973, 326 pp.*

No. 2 A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language, Jerry R. Hobbs, *1974, 266 pp.*

No. 3 Type Determination for Very High Level Languages, Aaron M. Tenenbaum, *1974, 171 pp.*

No. 4 A Comprehensive Survey of Parsing Algorithms for Programming Languages, Phillip Owens, *Forthcoming.*

A catalog of SETL Newsletters and other SETL-related material is also available. Courant Computer Science Reports are available upon request. Prepayment is required for Courant Computer Science Notes. Please address all communications to

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, N. Y. 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science NSO-3

TYPE DETERMINATION FOR VERY HIGH LEVEL LANGUAGES

Aaron M. Tenenbaum

Report No. NSO-3 prepared under
Grant No. NSF-GJ-1202X3 from
the National Science Foundation

Table of Contents

Abstract

Many very high level languages are declaration-free so that an identifier appearing in a program in such a language need not refer to objects of specific data type. Moreover, the semantic meaning of an operator symbol may depend on the dynamic types of its operands. This may cause such languages to be interpreted rather than compiled. In the present thesis, we present two methods which alleviate this problem by allowing the compile time detection of runtime types. The first method determines the types of runtime objects from the way these objects are defined, while the second determines the types of runtime objects from the way in which they are used. A SETL version of an implemented typefinder incorporating both methods is presented. Examples of the information developed by applying the typefinder to a few programs are also presented. Realization of the typefinder in languages of lower level than SETL is discussed. Several possible applications of the information developed by the typefinder are mentioned.

CHAPTER I

A TYPE DETERMINATION ALGORITHM

1. Introduction.

In many very high level languages, identifiers are not constrained to refer to objects of particular type. Also, in order to make maximum use of a small syntactic space, the meaning of operator symbols may depend on the types of their operands. In the SETL language (Schwartz 1973 a,b), for example, the plus sign represents addition for numeric operands, concatenation for tuples, character strings, or bit strings and union for sets. Such operations must either be interpreted directly or must compile into a series of subroutine calls. These subroutines must determine their operand types before executing the instructions which will perform the particular operation.

It is desirable to perform this type determination during compilation in order to speed execution. Such compile time type determination also makes it unnecessary to re-evaluate the type of an operand each time that it is encountered in a loop. For all but the shortest running programs, the time spent during compilation to determine the types of operands should therefore be less than the total time which would have to be spent on the same task during execution. Also, in production environments, the object code of a single compilation may be executed numerous times.

2. A Lattice of Types

In this section, we shall present the representation of the types of SETL run time objects which our algorithm will use. The representation is a simple one and similar representations are directly applicable to any high level programming language.

We define a family of *type symbols* each of which designates a "state of knowledge" about the type of a run time SETL object. These type symbols are composed from *elementary type symbols* which designate the basic scalar types of the language. More complex type symbols are developed from the elementary type symbols by using "constructor signs" which designate aggregates of elementary type objects or which indicate that our state of knowledge of the type of a specific object is not sufficient to identify a definite type but only enough to state that the object must be one of several alternate types.

We now present a formal definition of a family of type symbols for the SETL language. As an auxiliary quantity needed in the definition, we also define a function *nl*, called the *nesting level function* from the set of type symbols onto the set {0,1,2,3}. To simplify later proofs we also define a function *rl*, the *recursion level function* from the set of types into the set of nonnegative integers.

Definition 1. The following symbols are *elementary type symbols*, each designating a specific SETL type:

type symbol	type
ti	integer
tc	character string
tb	bit string
tn	null set
tt	null tuple
tu	the undefined atom
ta	the blank atom

We also define two more elementary type symbols which, unlike those of Definition 1, do not designate atomic types of the SETL language. These symbols designate two possible states of knowledge about a runtime object which play an important role in our considerations but which cannot be expressed in terms of the other elementary types.

Definition 2. The symbols *tz* and *tg* are elementary type symbols, and together with the seven elementary symbols of Definition 1, they form the totality of elementary type symbols.

The symbol *tz* designates "the error type" and indicates that a runtime object has not yet been defined or that it has been used in ways that may be seen to be incompatible.

Examples of this are identifiers which are used before they are defined or objects which are used both as integers and as bit strings without any conversions or overlay definitions having been encountered.

The symbol *tg* designates "the general type" and indicates that nothing is known about an object's type and that the object may be of any type. This symbol is also used when the type of an object is sufficiently complex that we do not wish to pay the overhead of keeping all of the detailed information about the object's type. This notion will be further elucidated in later definitions.

If *t* is an elementary type symbol then $r\ell(t) \equiv n\ell(t) \equiv 0$.

Next, we define a form of "compound" type symbol which will be used when we know that an object is of either one of two types but do not know its specific type.

Definition 3. Let *t1* be any type symbol not equal to *tz* or *tg* and let *t2* be any elementary type symbol not equal to *tz* or *tg*. Then the *alternation* of *t1* and *t2* is the symbol denoted by *t1|t2*, or *t2|t1*. The *alternands* of *t1|t2* are *t1*, *t2*, *t1|t2* and, if *t1* is itself an alternation, all the alternands of *t1*. The symbols denoted by *(t1|t2)|t3* and *t1|(t2|t3)* are identified with each other and will henceforth be denoted by *t1|t2|t3*. An alternation is a type symbol if no two of its alternands are the same. The *primary alternands of*

t1|*t2*|...|*tn*, where no *ti* is an alternation, are defined as *t1*,*t2*,...,*tn*. If *t1* is the non-elementary primary alternand of *t1*|*t2* then we define nℓ(*t1*|*t2*) ≡ nℓ(*t1*) and rℓ(*t1*|*t2*) ≡ rℓ(*t1*) + 1.

There are several points to note about the above definition. First, no alternation can have more than one non-elementary primary alternand. That is, we provide no way of designating an object which is known to be either a set of integers or a tuple of integers. Such an object will therefore be designated by *tg*. This is an arbitrary decision, and reflects a willingness to forego certain type information in order to simplify the storage and manipulation of type symbols. Where to draw the line between simplicity (which translates into efficiency of the algorithm) and the desire for more information is the question that must be faced in such decisions. An extremely simple, but almost useless, type system would be one in which alternations would be nonexistent and where any imprecision in the knowledge of types would result in the appearance of the indefinite type symbol, *tg*.

We do not define the alternations of *tg* and *tz* with any other types. If an object is known to be either of general type or an integer, for example, then that object could be of any possible type, so it is of type *tg*.

tg is thus an absorptive element for alternation. Similarly, *tz* is an identity element for alternation since if an object is known to be either erroneous or an integer, then it is known to be an integer. This follows from the simple fact that, in compilation, program correctness will be assumed unless we can prove otherwise.

The third and final point is that the syntactic meta-operation designated by "|" is both commutative and associative by the definition. Thus the forms t1|t2 and t2|t1 designate the same type symbol and the form t1|t2|t3 is well defined.

Example. ti|tb|tc designates "either an integer or a bit string or a character string". nℓ(ti|tb|tc) = 0 and rℓ(ti|tb|tc) = 2.

We now define a new class of type symbols which designate aggregate types.

Definition 4. Let *t1* be a type symbol and nℓ(*t1*) < 3. Then {*t1*} is a type symbol which designates "a non-null set of objects of type designated by t1." The nesting and recursion levels of such a symbol are defined by nℓ({*t1*}) ≡ nℓ(*t1*) + 1 and rℓ({*t1*}) ≡ rℓ(*t1*) + 1.

Examples. $\{\{ti\}\}$ designates a "non-null set of non-null sets of integers". $\{tb\}|tn$ designates a "possibly null set of bit strings." $\{ti|tc\}$ designates "a set each of whose elements is an integer or character string." $n\ell(\{\{ti\}\}) = 2$, $r\ell(\{\{ti\}\}) = 2$, $n\ell(\{tb\}|tn) = 1$, $r\ell(\{tb\}|tn) = 2$, $n\ell(\{ti|tc\}) = 1$, $r\ell(\{ti|tc\}) = 2$.

Definition 5. Let $t1$ be a type symbol and $n\ell(t1) < 3$. Then $[t1]$ is a type symbol which designates "a non-null tuple whose components are of type designated by $t1$". The nesting and recursion levels of such a symbol are defined by $n\ell([t1]) \equiv n\ell(t1) + 1$ and $r\ell([t1]) \equiv r\ell(t1) + 1$.

Examples. $[\{tb\}|ti|tc]$ designates "a non-null tuple, each of whose components is a set of bit strings, an integer or a character string". $n\ell([\{tb\}|ti|tc]) = 2$ and $r\ell([\{tb\}|ti|tc]) = 4$.

Definition 6. Let $t1, t2, \ldots, tk$ be type symbols such that $n\ell(t1) < 3$, $n\ell(t2) < 3$, ..., $n\ell(tk) < 3$. Then $<t1, t2, \ldots, tk>$ is a type symbol which designates "a tuple of length k whose components are of types designated by $t1, \ldots, tk$ respectively." The nesting and recursion levels are defined by $n\ell(<t1, \ldots, tk>) \equiv \max(n\ell(t1), \ldots, n\ell(tk)) + 1$ and $r\ell(<t1, \ldots, tk>) \equiv \max(r\ell(t1), \ldots, r\ell(tk)) + 1$.

Example. $<ti|tb,tc,\{tg\},tt>$ designates "a tuple of length 4 whose first component is an integer or a bit string, whose second component is a character string, whose third component is a non-null set and whose fourth component is the null tuple." $n\ell(<ti|tb,tc,\{tg\},tt>) = 2$, $r\ell(<ti|tb,tc,\{tg\},tt>) = 2$.

We do not define any type symbol with nesting level greater than 3. A smaller or larger nesting level limit would have been possible and would correspondingly decrease or increase the complexity of the type system. However a finite upper bound on the maximum level of nesting is necessary to insure that our algorithm will always terminate. We shall return to this point in our later discussion. Thus, an object known to be a set of sets of sets of integers will be designated as type $\{\{\{tg\}\}\}$ rather than $\{\{\{\{ti\}\}\}\}$, which is not a type symbol according to our definition. Similarly, $<ti,\{tb|[tc|<ti,tb,tc>]\},[[ti]],\{tc|[[tb]]\}>$ is designated by $<ti,\{tb|[tg]\},[[ti]],\{tc|[tg]\}>$.

Thus far we have defined a family of type symbols; we now build an algebraic structure on this family. We define two binary operations on the collection of type symbols and show that the collection together with the two operations forms a lattice.

The first operation, *type disjunction* is a generalization of alternation. Whereas alternation was a method for creating new type symbols, disjunction is an operation which operates on two types symbols to produce a third symbol. Recall that one of the primary alternands in any alternation must be an elementary type symbol. Also, tz and tg may not appear as alternands. Disjunction removes these restrictions. The disjunction of any two type symbols is that type symbol which designates an object which is known to be of either of the two input types. Note that the disjunction of two type symbols is one of the type symbols previously defined in Definitions 1-6 and is not some new "disjunct" type symbol.

We now present a formal definition of disjunction.

Definition 7. Given two type symbols, $t1$ and $t2$, we define the *type disjunction* of $t1$ and $t2$, written $dis(t1,t2)$, as follows:

$dis(t1,tg) \equiv dis(tg,t1) \equiv tg$ } for any
$dis(t1,t1) \equiv dis(t1,tz)$ } type symbol, $t1$
$\equiv dis(tz,t1) \equiv t1$ }

$dis(t1,t2) \equiv dis(t2,t1) \equiv t1|t2$ for any type symbol $t1$ which is not an alternation, $t1 \neq tz,tg$; and $t2$ such that $n\ell(t2) = 0$, $t2 \neq tz,tg,t1$.

$dis(t1|t2,t3) \equiv dis(t3,t1|t2)$ for any type symbols $t1,t2,t3$;
$\equiv t1|t2$ if $t3$ is an alternand of $t1|t2$.

$dis(t1\|t2,t3) \equiv dis(t3,t1\|t2) \equiv dis(t1,t3)\|t2$	for any elementary type symbol $t2$, and $t3$ any type symbol which is neither an alternation nor an alternand of $t1\|t2$ and is not equal to tz or tg; such that $dis(t1,t3) \neq tg$.
$dis(t1\|t2,t3\|t4) \equiv dis(dis(t1\|t2,t3),t4)$	for any type symbols $t1,t2,t3,t4$.
$dis(\{t1\},\{t2\})=\{dis(t1,t2)\}$ $dis([t1],[t2])=[dis(t1,t2)]$	for any type symbols $t1,t2$.
$dis(<t1,...,tk>,<t1',...,tk'>) \equiv <dis(t1,t1'),...,dis(tk,tk')>$	for any type symbols $t1,...tk,t1',...tk'$
$dis(<t1,...,tk>,<t1',...,tj'> \equiv [[dis: t \in <t1,...,tk, t1',...,tj'>]t]$	for any type symbols $t1,...,tk, t1',...,tj', j \neq k$, where $[dis: t \in <t1,...,tn>]t$ represents $dis(tn, dis(t(n-1),...,dis(t2,t1)...))$
$dis([t1],<t2,...,tk>) \equiv dis(<t2,...,tk>,[t1]) \equiv [[dis: t \in <t1,...,tk>]t]$	for any type symbols $t1,...,tk$.
$dis(t1,t2) \equiv tg$	for all type symbols $t1,t2$ which do not meet the above conditions

Examples

dis(*ti*,*tb*) = *ti*|*tb*

dis({*ti*},*tb*) = {*ti*}|*tb*

dis(*ti*|*tb*,*tb*|*tc*)= *ti*|*tc*|*tb*

dis({*ti*}|*tb*,*tg*) = *tg*

dis([*ta*],<*ti*,*tb*,*tc*>)=[*ta*|*tb*|*tc*|*ti*]

dis({*ti*}|*tn*,[*ti*]|*ti*) = *tg*

dis({[*tb*]|*ti*},{<[*tc*],*ti*>|*ta*}) = {[[*tc*]|*tb*|*ti*]|*ta*|*ti*}

dis(*ti*|*tc*,*tb*) = *ti*|*tb*|*tc*

dis({*ti*}|{*tb*})= {*ti*|*tb*}

dis({*ti*|*tb*},{*tb*|*tc*})={*ti*|*tc*|*tb*}

dis({*ti*}|*tb*,{*tc*}|*ti*)
= {*ti*|*tc*}|*ti*|*tb*

dis([*ti*],{*ti*}) = *tg*

Disjunction is both commutative and associative. This can be proven by induction on the recursion level using a case-by-case analysis. Intuitively, the disjunction of two type symbols is that type symbol which designates an object which is known to have either of the two types designated by the input symbols, so that the order of the operands is immaterial. Similarly the disjunction of three type symbols is that type symbol which designates an object known to have one of the three types designated by the input symbol so that it makes no difference which two type symbols we choose to disjoin first.

The class of objects of type designated by the disjunction of two types *t1* and *t2* includes both the class of objects of the type designated by *t1* and the class of objects of the type designated by *t2*. This is because in forming the disjunction *t* from *t1* and *t2* we are becoming "vaguer" in the information that is known

about an object's type, so that more objects may be described by *t* than by either *t1* or *t2*. The type *tg*, which is the absorptive element for disjunction (as it was for alternation), can designate any SETL runtime object.

We now define a second binary operation on the set of type symbols. This operation, *type conjunction*, designates the type of an object known to be of both of the types designated by its two operands, or the error type symbol, *tz*, if the two types are incompatible.

Definition 8. Given the two type symbols *t1* and *t2*, we define the *type conjunction* of *t1* and *t2*, written *con(t1,t2)* as follows:

con(*t1*,*tg*) ≡ con(*tg*,*t1*) for any type symbol *t1*.

≡ con(*t1*,*t1*) ≡ *t1*

con(*t1*,*tz*) ≡ con(*tz*,*t1*) ≡ *tz*

con({*t1*},{*t2*}) ≡ {con(*t1*,*t2*)}

con([*t1*],[*t2*]) [con(*t1*,*t2*)] } for any type symbols *t1*,*t2* such that con(*t1*,*t2*) ≠ *tz*.

con(<*t1*,...,*tk*>,<*t1'*,...,*tk'*>) ≡ <con(*t1*,*t1'*),...,con(*tk*,*tk'*)>

for all type symbols *t1*,...,*tk*, *t1'*,...,*tk'*,

provided that *1* ≤ ∀*j* ≤ *k* , con(*tj*,*tj'*) ≠ *tz* .

con[[*t1*],<*t2*,...,*tk*>) ≡ <con(*t1*,*t2*),con(*t1*,*t3*),...,con(*t1*,*tk*)>

for all type symbols *t1*,...,*tk*

provided that 2 ≤ ∀*j* ≤ *k* , con(*t1*,*tj*) ≠ *tz*.

con(*t1*|*t2*,*t3*) ≡ dis(con(*t1*,*t3*),con(*t2*,*t3*))

for all type symbols *t1*,*t2*,*t3*.

con(*t3*,*t1*|*t2*) ≡ dis(con(*t1*,*t3*), con(*t2*,*t3*))

for all type symbols *t1*,*t2*,*t3*; *t3* not an alternation

con(*t1*,*t2*) ≡ tz

for all type symbols *t1*,*t2* such that none of the above conditions is met.

Examples

con({*ti*}|*tb*, *tc*|*tb*) = *tb*

con(<*ti*,*tc*>,<*ti*,*tc*,*ta*>) = *tz*

con({*ti*}| *ti* ,{*tb*|*tc*}) = *tz*

con({*ti*|*tc*},{*tb*|*ti*}) = {*ti*}

con([*ti*|*tb*],<*ti*,*ti*,*ti*,*tb*>)
= <*ti*,*ti*,*ti*,*tb*>

con({*ti*}|[*tc*],[*tc*|*tb*]) =[*tc*] .

Conjunction is also commutative and associative. The proof is by induction on the recursion level, using a case-by-case analysis. The class of objects whose types may be designated by the conjunction of two types is included in the class of objects whose types may be designated by either of the input types. This is because in forming the conjunction *t* of *t1* and *t2* we are becoming "more precise" in the information that is known about an object's type so that fewer objects are described by *t* than by either *t1* or *t2*. The type *tz* which is the absorptive element for conjunction designates no actual SETL runtime object, and when it appears indicates that no object with the properties required in some particular program context can exist.

Two more facts can be proven about the set T of type

symbols and the operations of disjunction and conjunction. These are the *absorptive laws* and state that for all type symbols *t1* and *t2*,

$$con(t1, dis(t1,t2)) = t1 \qquad dis(t1, con(t1,t2)) = t1.$$

Again, the proofs are by induction on the recursion levels.

The absorptive laws together with the commutative and associative laws are all that is necessary to show that the system (T,con,dis) forms a lattice, so that we may now use all of the properties of a general lattice in connection with our type system. In particular, if we define the relation $\leq$ on T by $t1 \leq t2$ if and only if $dis(t1,t2) = t2$ (this is of course completely equivalent to $con(t1,t2) = t1$), then the system $\{T, \leq\}$ forms a partially ordered set. Informally, the statement $t1 \leq t2$ expresses the notion that a larger class of objects can be designated by type symbol *t2* than by *t1*, i.e., that the type information implied by *t2* is "vaguer" than that implied by *t1*.

tg is the "vaguest" element of T and can designate any SETL run time object, while *tz* is the "most precise". element of T and can designate no SETL runtime object.

Definition 9. Given a partially ordered set $\{P, \leq\}$ and $S \subseteq P$. Then a *maximal* [*minimal*] element of S is an element $p \in S$ such that $(\nexists\, p' \in S - \{p\})$, $p' \geq p$ $[p' \leq p]$. A *maximum* [*minimum*] element of S is an element $p \in S$ such that $(\forall p' \in S)$, $p \geq p'$ $[p \leq p']$. A lattice is called

bounded if it has a maximum and minimum element. The maximum element is denoted by 1 and the minimum element by 0.

Theorem 1. {T,dis,con} is a bounded lattice with maximum element *tg* and minimum element *tz*.

Proof: Follows immediately from the identities

dis(*t1*,*tg*) = *tg* and dis(*t1*,*tz*) = *t1*.

Definition 10. Given a partially ordered set {P, $\leq$} and S = $\{p_1, p_2, \ldots\} \subseteq P$, then S is a *chain* if $p_1 \leq p_2 \leq p_3 \leq \ldots$. The *length* of S is its cardinality.

Despite the fact that we have imposed a limit on the nesting level of type symbols, the set T is not a finite set. This is because of the presence in T of symbols representing tuples of arbitrarily large known length. However, we can prove that any chain in T must be of finite length.

Theorem 2. Any chain in T is of finite length.

Proof: Suppose that a chain of infinite length existed in T. Then there would exist a chain containing two elements <tl,...,tk> and <tl',...,tj'> where k $\neq$ j. Then <tl,...,tk> $\leq$ <tl',...,tj'> which means that

dis(<tl,...,tk>,<tl',...,tj'>) = <tl',...,tj'>.

But this is clearly not the case since

dis(<tl,...,tk>,<tl',...,tj'>)=[[dis:t∈<tl,...,tk,tl',...,tj'>]t].

We will see later that the above fact that all chains are of finite length is of importance in insuring termination of the algorithms to be presented. Note also that if we had not imposed a maximum nesting level this property would fail to hold and we could have infinite chains such as

$$tg \geq \{tg\} \geq \{\{tg\}\} \geq \{\{\{tg\}\}\} \geq \{\{\{\{tg\}\}\}\} \geq \ldots$$

3. An Intuitive Introduction to Type Determination.

Having presented a method for representing data types and having proved some elementary properties of the resultant structure, we shall now give an intuitive description of the method we propose to use for determining the types of runtime objects from a static program text. There are two methods which will be used to determine object types. The first method examines the way in which objects are defined, and determine the data types which can result from particular methods of definition. Initially the types of all constants in a program are known and all program variables (with the exception of those which are defined by read statements)are ultimately defined in terms of these constants. If an object O is defined by applying an operator to two constants, the type of O can be deduced from knowledge of the operator and the data types of the constants. Once the type of this defined object has been deduced, it can be used in deducing the type of other objects which are defined in terms of it.

Consider, for example, the following SETL sequence:

(1)

```
x = {y};
z = x + {1,2};
```

Although nothing is known about the type of *y* (here, we ignore the fact that the type of *y* may be known from the way it was defined in a previous program segment) and its type must therefore be taken as *tg*, it can be deduced that *z* is a set (described by {*tg*}). We also see that in the operation z = x + {1,2}, the plus operator must represent set union; so that during compilation we can generate "union-forming" code directly rather than having to call a subroutine which must first make a runtime determination of the meaning of the plus sign.

This first method of type determination propagates type information in the direction of execution flow. The types of quantities which serve as inputs to operations determine the type of the operation output and thus contribute to the determination of the types of other quantities. This method applies in straightforward fashion to the case of a *basic block*, i.e. a piece of straight line code with a single entry and exit point. If a definition appears within such a block, the operation which defines the input to that definition is readily identifiable as the last operation whose target identifier is the same as the input identifier.

However, in the presence of flow, things become more complicated. Consider, for example, the situation shown in Figure 1.

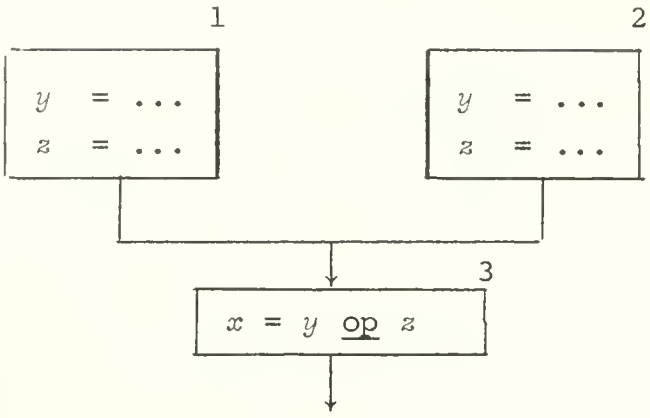


Figure 1. The Convergence of Two Execution Paths.

In determining the type of the object referred to by x in block 3, one must take into account the possibility that during execution, control may reach block 3 from either one of blocks 1 and 2. In general, the types of the objects referred to by y and z will differ depending on which block is actually executed immediately before block 3. Thus at compile time one only knows that the objects referred to by y and z in block 3 are describable by the disjunctions of the type symbols which describe these same identifiers in blocks 1 and 2. If in block 1 y were known to be of type ti, and in block 2, y were known to be of type tb, then in block 3, y is known to be of type $tb|ti$.

Another complication arises in the case of a loop as in Figure 2.

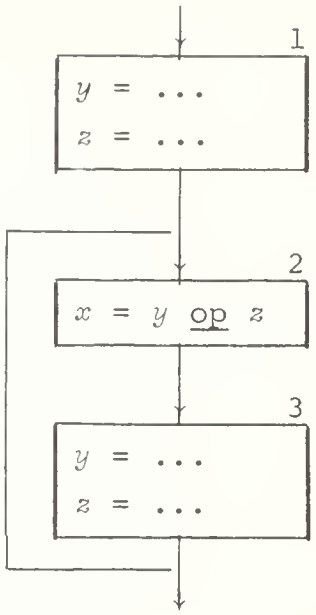


Figure 2. A loop complicating the first method of type determination.

Proceeding in order of control flow, we can determine the types of y and z in block 1, the type of x in block 2 and the types of y and z in block 3. However, we must then re-evaluate the type of x in block 2 since control may reach block 2 from block 3 rather than from block 1. If the definitions of y and z in block 3 depend on x in turn, then repeated re-evaluations of the types of x, y and z in blocks 2 and 3 are necessary until the type information stabilizes. In fact, it must be proved that stabilization is assured in a finite number of steps. We will address ourselves to this question later in the discussion.

The second or "backwards" method of type determination deduces the types of runtime objects from the way in which these objects are used. The application of certain operations to an object can serve to restrict the range of types which that object may validly take on. For example, consider the SETL sequence:

(2)
```
1.  read y,z;
2.  x = y + z;
3.  w = hd x;
```

In statement 3, the fact that the hd operation is applied to x implies that x is a tuple. We then note that the plus operation in statement 2 produces a tuple which implies that its inputs are tuples and that the plus operation represents tuple concatenation. This allows us to deduce the types of y and z which are read at statement 1.

This method of type determination proceeds in reverse order of control flow. Quantities $Q_1, Q_2, \ldots$ have their types determined from the way that they are subsequently used. This information may then assist in determining the types of quantities which are used in defining $Q_1, Q_2, \ldots$.

Whereas the first method provides no information about an object defined by a read statement, the second method may allow us to deduce the type of such an object from its subsequent uses. This allows us to insert runtime checks at each read statement to ensure that the item read in is of a type which is compatible with subsequent uses.

As with the first method, the presence of flow complicates the second method. Consider the example of Figure 3:

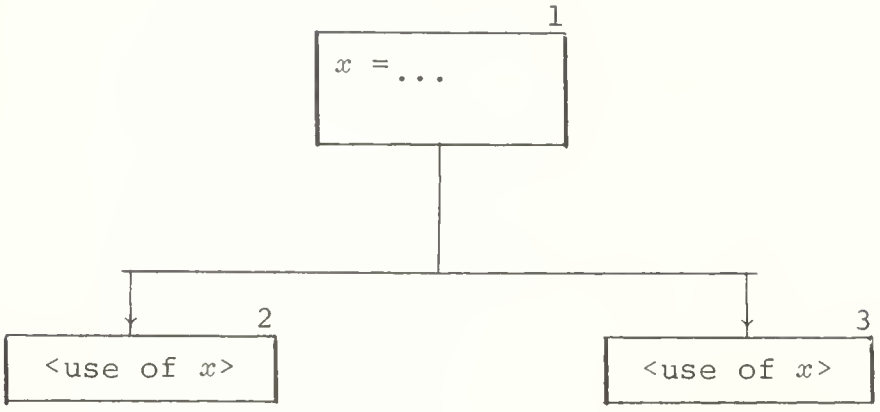


Figure 3: Diverging Execution Paths.

The runtime decision as to whether to enter block 2 or block 3 from block 1 may depend on the type of the object referred to by x at the time of the decision. Thus, one of the uses may be inapplicable to an object of the type of x, but a dynamic runtime check can avoid the inapplicable use. It may be that, if the code fragment in Figure 3 is embedded in a loop, different paths may be taken at different iterations of the loop. The most that can be said for the type of x, therefore, is that it is known to be describable by the disjunction of the two types indicated by the two uses. Further, if block 1 in Figure 3 were a program exit no inference could be made from the uses of x in blocks 2 and 3 about the type of x in block 1. This is because a type-dependent runtime decision to exit the program might be made.

In such a case, we can only assign the type *tg* to x.

A similar situation is shown in Figure 4.

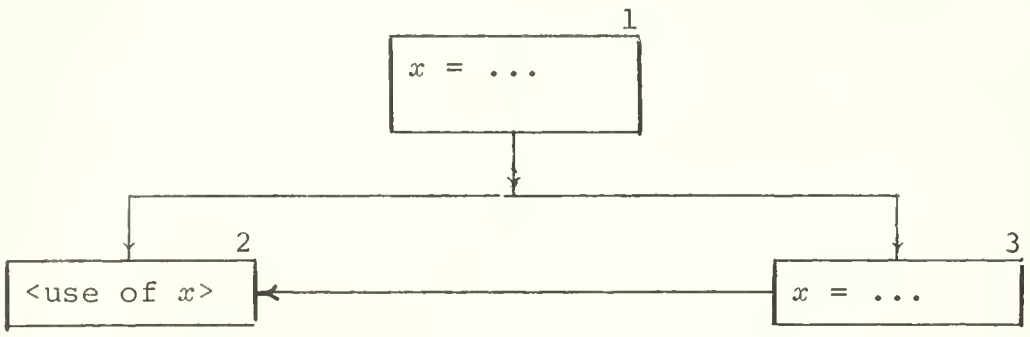


Figure 4. A Use Lying on a Path Past a Redefinition.

The use of x in block 2 may act as the determinant of the type of x in block 3 but not of the type of x in block 1. This is because runtime control may enter block 2 or 3 from block 1 depending on the type of x. More specifically, control might be routed to block 3 in order to first redefine x so that it will conform to the use in block 2. This illustrates the general fact that no use of a variable which appears on a path past a redefinition is valid in determining the type of an earlier definition.

As in the first method, if a loop occurs we may be unable to use strictly (reverse) sequential processing in the second method. Consider the example of Figure 5.

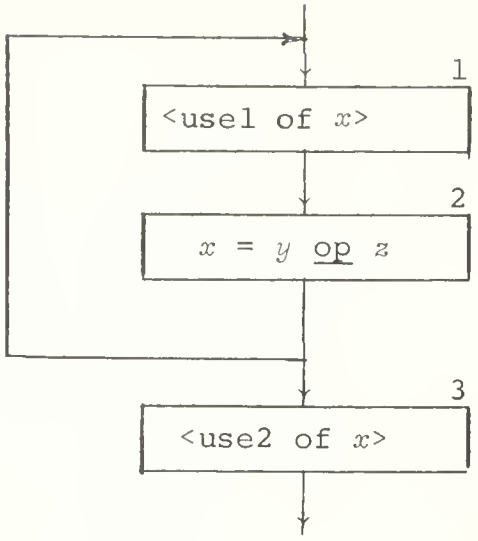


Figure 5. A Loop Complicating the Second Method of Type Determination.

Proceeding sequentially in reverse execution order, we may determine the type of <use2 of x> which may allow us to determine the types of y and z in block 2. However after determining the type of <use1 of x> we must re-evaluate the types of y and z to reflect the new type possibilities for x. If <use1 of x> is in a definition of y or z then repeated re-evaluations of the types of x, y and z in blocks 1 and 2 are necessary until the situation stabilizes. Again, as in the first method, it must be proved that stabilization is assured in a finite number of steps.

We shall shortly construct a formalism to express the above notions more exactly. This formalism will be used both to define a type determination algorithm and to prove the correctness of that algorithm.

4. Program Representation and Use-Definition Chaining.

In this section we shall describe a schematized form in which programs will be represented in our type determination algorithm. Each *program* is regarded as a set of *basic blocks* together with a multivalued *successor function* defined on the set of basic blocks and with range in that same set. Each basic block consists of an ordered set of *definitions*. Within each block, control is assumed to proceed from the first definition in the block, through all succeeding definitions in the block according to the ordering. The possible paths of control flow between blocks of a program are defined by the block successor function. Each definition in a schematized program consists of three components:

i) an operator, *op*;

ii) a set of input variables, $v_1, \ldots, v_k$;

iii) an output variable, x.

Let *defs* be the set of definitions in a program, ordered in some arbitrary fashion. We let d_i represent the ith definition in *defs*. If $d = d_i$, we let *type(d)* and *type(i)* denote the type of the output variable of definition d.

Note that the same variable may be the target of different definitions each of which produces a result of different type. It is for this reason that we choose to have our algorithm associate type symbols with the output of specific definitions rather than with variables. We

also distinguish between a variable v_i and a use of that variable at a specific point in our schematized program. If u refers to a specific use of a variable v, we let $type(u)$ denote the type of the variable v.

Given a use u of a variable v and a definition df of that same variable, u can access the value stored by df if and only if there is a path from the program entry to u on which df is the last definition of v. If this is the case, we say that u and df are *chained* to each other. Note that on any path to a variable use there can be at most one definition which stores a value that the use is able to access.

The type of a variable at a point of use at a specific instant during runtime is the type of one of the objects stored by the definitions which are chained to it. The specific object used is of course dependent on the path actually taken to the use during runtime. However we can assert that the type of a variable at a point of use at any runtime instant is (in the sense of the type lattice) less than the disjunction of the types of all definitions chained to that use.

We now establish a few notations which allow the above relations to be expressed formally and conveniently. Given a use u, let $ud(u)$ be the set of definitions which can create values used at u; i.e., the set of all definitions chained to u. This function can be calculated efficiently

by employing the interval based use-definition chaining process originally developed by John Cocke and described in (Schwartz, 1973b).

Following a notational convention of SETL, we let $[dis:\ t \in T]t$ represent the disjunction of all type symbols in a set T. In terms of this notation, the relation-relationships described informally above may be written as

(3) $\quad type(u) \leq [dis:\ df \in ud(u)]type(df)$

for all uses u in the program. It is understood that if $ud(u)$ is the null set, then the right side reduces to tz. We shall now begin to describe a formal system of relations between object types and SETL operators.

5. A Type Calculus and the First Method of Type Determination.

Recall that the first method of type determination seeks to determine the type of an output variable from the way in which the variable is defined. To specify this method, we use a *type calculus* which defines the way in which input types are operated on by the operations in a programming language to produce output types. For example, the set of rules applying the plus operator in SETL is as follows:

t1+t1 = t1 for all elementary types t1 ≠ ta, tu;

t1+t2 = tz for all types t1 = tz, ta, tu, and t2;

tg+t1 = t1 for t1 = ti,tb,tc,tg;

t1+t2 = tz for t1 = ti,tb,tc and t2 ≠ t1,tg;

t1+t2 = tz for t1 = tn, tt and t2 ≠ t1, tg, elementary;

t1+(t2|t3) = dis(t1+t2,t1+t3);

tt+<t1,...,tk> = <t1,...,tk>;

tg+<t1,...,tk> = tg+[t1] = [tg];

tt+[t1] = [t1];

tt+{t1} = tn+[t1] =

= tn+<t1,...,tk> = tz;

tn + {t1} = {t1};

tg + {t1} = {tg};

{t1}+{t2} = {dis(t1,t2)};

[t1]+[t2] = [dis(t1,t2)];

<t1,...,tj>+<t(j+1),...,tk>

= <t1,...,tk>;

<t1,...,t(k-1)>+{tk}

= [t1] + {t2} = tz;

[t1]+<t2,...,tk>

= [[dis:t∈<t1,...,tk>]t];

tt+tg = [tg]|tt;

tn+tg = {tg}|tn .

} for all types t1,...,tk

Since the plus operator is always commutative, the order of operands in the above equations is irrelevant.

Similar rules may be stated for all other operators of SETL, and more generally for operators of other high level languages.

Consider a particular definition $d = \langle op, v_1, \ldots, v_k, x \rangle$. We will write $calc(op, t1, \ldots, tk)$ to denote the type symbol which results when an operator op acts on a set of type symbol inputs, $t1, \ldots, tk$. Let τ_1 indicate the instant before the execution of definition d, and τ_2 the instant after. Let $u_1, \ldots, u_k$ represent the uses of variables $v_1, \ldots, v_k$ in definition d. Let $type_1(u_i)$ represent the type of the object denoted by v_i at time τ_1 and $type_2(u_i)$ represent its type at time τ_2. Similarly, let $type_1(d)$ and $type_2(d)$ be the types of the object denoted by x at times τ_1 and τ_2. Then clearly,

(4) $type_2(d) \leq calc(op, type_1(u_1), \ldots, type_1(u_k))$.

Note that the $calc$ function is monotone increasing (in the sense of our type lattice) as a function of $t1, \ldots, tk$. This is clear, since if an input variable may take on a larger range of types, the output variable may also take on a correspondingly larger range of types.

Since $calc$ is monotone we may substitute inequality (3) into (4) to produce

(5) $type_2(d) \leq calc(op, [dis: df \in ud(u_1)]\, type_1(df), \ldots,$
$\ldots\ [dis: df \in ud(u_k)]\, type_1(df))$.

Definition 11. Given a program containing n definitions, and given a specific instant during the execution of the program; let $x_i = type(d_i)$ at that particular instant. Then the *type status* of the program at that instant is the vector $(x_1,\ldots,x_n)$.

At the start of execution, all variables are undefined; no definitions having been executed. Therefore the type status of the program at the start of execution is $(tz,tz,\ldots,tz)$. If $df = d_i$, let us denote x_i by x_{df} and define

(6) $$forward(d) \equiv calc(op,[dis: df\in ud(u_1)]x_{df}, \ldots \ldots [dis: df \in ud(u_k)]\ x_{df})$$

which is a function of $x_1,\ldots,x_n$ and may therefore be written

(7) $$forward(d_i) = f_i(x_1,\ldots,x_n)\ .$$

Note that each of the functions f_i for $1 \leq i \leq n$ is monotone increasing. If at some instant in the program execution, the type status is $(x_1,\ldots,x_i,\ldots,x_n)$ then the execution of definition d_i transforms the type status to $(x_1,\ldots,x_i',\ldots,x_n)$ where $x_i' \leq f_i(x_1,\ldots,x_n)$. This is an immediate consequence of (5)-(7).

In assigning a type to a definition d by a compile time analysis, we must be sure that the type assigned to d is greater than or equal to the disjunction of the types which the output variable of d actually takes on during execution. The preceding discussion shows us how to obtain such an "upper type bound." Informally, an upper bound on the disjunction of the types which the variables of a program may taken on during runtime is the maximum type status to which $(tz,\ldots,tz)$ is transformable by the functions f_i.

In order to develop an algorithm for type determination, we are therefore motivated to study monotone functions on partially ordered sets.

6. Functions and Transformaions on Partially Ordered Sets.

We assume a knowledge of the definition of a *partially ordered set* and of a *monotone increasing* (which will hereafter be referred to as a *monotone*) function on a partially ordered set. We will also refer to the terms defined in Definitions 9 and 10, above.

Theorem 3. Let P be a partially ordered set under a partial ordering $\leq$. Define a relation $\leq'$ on P^n by $(p_1,\ldots,p_n) \leq' (q_1,\ldots,q_n)$ if and only if $(\forall 1 \leq k \leq n) p_k \leq q_k$. Then $\{P^n, \leq'\}$ is a partial ordering. Moreover, if 0 is the minimum element of P_1 then $(0,\ldots,0)$ is the minimum

element of P^n; and if 1 is the maximum element of P, then (1,...,1) is the maximum element of P^n.

The proof of Theorem 3 is quite simple since the properties of $\leq'$ follow directly from the corresponding properties of $\leq$, and the proof will not be given here. We subsequently drop the "prime" from $\leq'$ and use $\leq$ for both the partial ordering in P and in P^n, it being clear by the context to which one we are referring.

Theorem 4. Let P be a partially ordered set all of whose chains are of finite length. Then all chains of P^n are also finite length.

Proof: Let $s_0 < s_1 < s_2 < \ldots$ be an infinite chain in P^n. Let s_i^j be the jth component of s_i. Then consider the n chains in P:

$$s_0^1 \leq s_1^1 \leq \ldots$$
$$s_0^2 \leq s_1^1 \leq \ldots$$
$$\vdots$$
$$s_0^n \leq s_1^n \leq \ldots$$

Since all of these chains are finite, $\exists k$ such that $\forall j$, $\forall m \geq k$, $s_m^j = s_{m+1}^j$. But then $\forall m \geq k$, $s_m = s_{m+1}$, so that the chain is finite.

Let P be a partially ordered set with minimum element O all of whose chains are finite. Let $\{f_1,\ldots,f_n\}$ be a set of monotone functions, and suppose we wish to solve the system of equations

$$
\begin{aligned}
x_1 &= f_1(x_1,\ldots,x_n) \\
&\vdots \\
x_n &= f_n(x_1,\ldots,x_n) \ .
\end{aligned}
\tag{8}
$$

In general more than one solution is possible. To find the minimum solution, consider the following sequence in P^n:

$$
\begin{aligned}
s_0 &= (x_1^0, x_2^0, \ldots, x_n^0) = (0,0,\ldots,0) \\
s_1 &= (x_1^1, x_2^1, \ldots, x_n^1) = (f_1(0,\ldots,0), f_2(0,\ldots,0) \ \ldots f_n(0,\ldots 0)) \\
&\vdots \\
s_{i+1} &= (x_1^{i+1}, x_2^{i+1}, \ldots, x_n^{i+1}) = (f_1(x_1^i, x_2^i, \ldots, x_n^i) \ , \\
&\qquad \ldots \ f_n(x_1^i, x_2^i, \ldots, x_n^i))
\end{aligned}
\tag{9}
$$

Theorem 5. There exists a $k > 0$ such that $s_k = s_{k+1}$ in (9).

Proof: If the theorem were false, then by the monotonicity of $f_k \ldots, f_n$ there would exist an infinite chain $s_0 \leq s_1 \leq \ldots$ in P^n which is impossible by Theorem 4. Q.E.D.

Definition 12. Given a sequence $\{s_0, s_1, \ldots\}$ in a partially ordered set P, the sequence *converges* to $p \in P$ if $\exists k \geq 0$ such that $(\forall i \geq k)$ $s_k = p$.

Corollary. The sequence $\{s_0, s_1, \ldots\}$ in (9) above converges to s_k where k is any integer such that $s_k = s_{k+1}$.

Theorem 6. s_k is a minimum solution of (8).

Proof: Let s be a solution to the system of equations (8). Then $s = (x_1, x_2, \ldots, x_n)$. Since $(0,\ldots,0)$ is the minimum element of P^n, $s \geq (0,\ldots,0) = s_0$. But if $s \geq s_j = (x_1^j, \ldots, x_n^j)$, then $x_i \geq x_i^j$ for all $0 \leq i \leq n$. But by the monotonicity of f_i, this implies that $x_i = f_i(x_1,\ldots,x_n) \geq f_i(x_1^j,\ldots,x_n^j)$ for all $1 \leq i \leq n$. Therefore, $s = (x_1,\ldots,x_n) \geq (f_1(x_1^j,\ldots,x_n^j),\ldots f_n(x_1^j,\ldots x_n^j))$ $= s_{j+1}$ so that $s \geq s_{j+1}$. By induction, $s \geq s_k$ which proves that s_k is the minimum solution. Q.E.D.

Now, let us examine the system of inequalities:

$$\begin{aligned} x_1 &\leq f_1(x_1,\ldots,x_n) \\ &\vdots \\ x_n &\leq f_n(x_1,\ldots,x_n) \end{aligned} \tag{10}$$

where $\{f_1,\ldots,f_n\}$ is a set of monotonic functions.

Theorem 7. If $s = (x_1,\ldots,x_n)$ is a solution to the system (10), then $s' = (f_1(x_1,\ldots,x_n),\ldots,f_n(x_1,\ldots,x_n))$ is also a solution and $s \leq s'$.

Proof: Clearly, since s is a solution,

$$s = (x_1,\ldots,x_n) \leq (f_1(x_1,\ldots,x_n),\ldots,f_n(x_1,\ldots,x_n)) = s' .$$

To see that s' is a solution, note that:

$$f_i(x_1,\ldots,x_n) \leq f_i(f_1(x_1,\ldots,x_n),\ldots,f_n(x_1,\ldots,x_n))$$

for all $1 \leq i \leq n$ by the monotonicity of $f_1,\ldots,f_n$. Q.E.D.

Corollary. A maximal solution to the system (10) must also satisfy the system (8).

Definition 13. Let S be a set. Then a *transformation* on S^n is a function from S^n into S^n. If $\hat{f}$ is a transformation on S^n, then $(x_1,\ldots,x_n)$ is said to be *transformable* into $\hat{f}(x_1,\ldots,x_n)$ by $\hat{f}$. If $T = \{\hat{f}_1,\ldots,\hat{f}_k\}$ is a set of transformations on S^n then $(x_1,\ldots,x_n) \in S^n$ is said to be *transformable* into $(y_1,\ldots,y_n) \in S^n$ by T if there exists a finite sequence $\hat{f}_{i_1},\ldots,\hat{f}_{i_j} \in T$ such that

$$(y_1,\ldots,y_n) = \hat{f}_{i_1} \cdots \hat{f}_{i_j}(x_1,\ldots,x_n)$$

Definition 14. Let f_i be a function from S^n into S. Then f_i is said to *induce the transformation* $\hat{f}_i$ which is the function from S^n into S^n defined by $\hat{f}_i(x_1,\ldots,x_i,\ldots,x_n) \equiv (x_1,\ldots,f_i(x_1,\ldots,x_n),\ldots,x_n)$.

Definition 15. An element $(x_1,\ldots,x_n)$ of S^n is *transformable* into $(y_1,\ldots,y_n) \in S^n$ by a set of functions (from S^n into S), $F = \{f_1,\ldots,f_k\}$, if $(x_1,\ldots,x_n)$ is transformable into $(y_1,\ldots,y_n)$ by the set of transformations induced by the elements of F; i.e., by the set $T = \{\hat{f}_1,\ldots,\hat{f}_k\}$.

Theorem 8. Let P be a partially ordered set with minimum element 0 and chains of finite length. Let $F = \{f_1,\ldots,f_n\}$ be a set of monotone functions from P^n into P and let $G_i = \{g_j^i$ a function from P^n into P $\mid$ $g_j^i \leq f_i\}$. Then the maximum element of P^n into which $(0,0,\ldots,0)$ is transform-

able by $G = \bigsqcup_{i=1}^{n} G_i$ is the element s_k to which the sequence $s_0, s_1, \ldots$ in (9) converges.

Proof: Let us first show that $(0,0,\ldots,0)$ is transformable into s_k by F. This follows from the fact that $s_0 = (0,\ldots,0)$ and $s_{i+1} \leq \hat{f}_1 \ldots \hat{f}_n(s_i)$. Therefore $s_k \leq (\hat{f}_1 \ldots \hat{f}_n)^k(0,\ldots,0)$. But $s_{i+1} \geq \hat{f}_j(s_i)$ for all $1 \leq j \leq n$. Since $s_k = s_{k+1} = s_{k+2} = \ldots$, we have that

$$(\hat{f}_1 \ldots \hat{f}_n)^k(0,\ldots,0) \leq s_{nk} = s_k \leq (\hat{f}_1 \ldots \hat{f}_n)^k(0,\ldots,0)$$

which gives us the desired result. Since $F \subseteq G$, $(0,0,\ldots,0)$ is transformable by G into s_k.

Let s be some other element to which $(0,\ldots,0)$ is transformable by G. Then

$$s = \hat{g}_{j_1}^{i_1} \hat{g}_{j_2}^{i_2} \ldots \hat{g}_{j_m}^{i_k}(0,\ldots,0) \quad \text{where} \quad g_{j_k}^{i_k} \in G_{i_k},$$

so that

$$s \leq \hat{f}_{i_1} \hat{f}_{i_2} \ldots \hat{f}_{i_n}(0,\ldots,0) \leq (\hat{f}_1 \ldots \hat{f}_n)^m(0,\ldots,0) \leq s_k. \quad \text{Q.E.D.}$$

The application of the above formalism to the first method of type determination is clear. If P is our type lattice then P^n is the set of all possible type status vectors which can describe a program of n definitions at any instant of its execution. If we let f_i be as defined in (7), then by (4), the execution of a particular definition d_i corresponds to the application of a particular $\hat{g}_j^i$ to the type status, where $g_j^i \leq f_i$. The compile time type

information which we are able to develop is defined by the maximum type status vector to which $(tz,\ldots,tz)$ is transformable by all possible such functions g. By Theorem 8 and its proof, this is given by the vector

(11) $$(x_1,\ldots,x_n) = (\hat{f}_1\ldots\hat{f}_n)^k(tz,\ldots,tz)$$

where k is the smallest integer such that $x_i = f_i(x_1,\ldots,x_n)$ for all $1 \leq i \leq n$.

This shows that a useful upper bound of the types which a variable appearing at a specific place in the program may take on is obtainable algorithmically. The algorithmic process just described is our first method of type determination.

7. The Second Method of Type Determination.

We now continue the heuristic discussion of the second method of type determination begun in Section 3 with a view to formalizing the results. Recall that the second method of type determination discovers the type of an object from the way in which it is subsequently used and assumes program correctness to the extent that objects of a specific type are not used in a way which is incompatible to that type.

Let u be a use of a variable x within a definition, df. By $backtype(u)$ we will denote the type of x as determined from the use u. The question of what

"environmental" influences will be used to determine *backtype(u)* is an important one. In the simplest system, only the operator appearing within *df* and the position of *u* as an operand of that operator will determine *backtype(u)*. For example, if *u* is an input to a hd operation, then *backtype(u) = tt|[tg]*; if it is the first argument of a npow operation (*int* npow *set* returns the subset of the power set of *set* whose elements have cardinality *int*) then *backtype(u) = ti;* if it is the second argument of an npow operation then *backtype(u) = tn|{tg}*. Such a system, however, will probably be too weak to produce any significant results. By the nature of very high level languages, the operator symbols which are most commonly used are those which apply to a wide variety of object types. In SETL, for example, almost nothing can be deduced from the fact that a use is an argument of a plus operation since a variable used in such a manner may be an integer, bit string, character string, set or tuple. Unless the type calculus were rich enough to provide a type symbol for "not a blank atom or the undefined atom" which would increase the complexity of the calculus greatly, such a use can only be identified as being of type *tg*.

For the above reasons, we have chosen to make use of all available information about the type of the output variable of *df* in determining *backtype(u)*. In the above example, if *u* appears in definition *df* and the output variable

of *df* is known to be of type *ti* (either by the first method of type determination, e.g. if *type(u)* = *tg* but the type of the companion argument to the plus operation is known to be of type *ti*; or by the second method from a later use of the output variable of *df*) then *backtype(u)* = *ti*. For an example showing how types could be propagated backwards using such a scheme, see (2) above and the discussion which follows it.

Another possibility is to use the types of any companion arguments which appear together with *u* as inputs For example, if *u* were "plussed" with a set we would know that *backtype(u)* = *tn*|{*tg*}. This possibility, however, will not be explored further. Our discussions will focus on the use of information concerning operators and the type of their output object to determine the type of their input objects.

Given a definition *d* and a block *b*, let *du(d,b)* be the set of all uses of the output variable of *d* which are chained to *d* and which appear in the block *b*.

Consider a basic block *b* which, for simplicity, we will assume does not appear in a program loop. That is, we assume that there is no possible execution path from the exit of *b* back to its entry. If it happened that all the uses to which definition *d* is chained appeared in the same basic block *b* in which *d* appeared, then the type ascribed to the output of definition *d* by our second

method of type determination would be the quantity *back(d)* defined by the equation:

(12) $back(d) \equiv [con:\ u \in du(d,b)]backtype(u)$.

The following argument justifies this equation:

(a) all the uses in *du(d,b)* and no others may employ the value stored by definition *d*; and

(b) *backtype(u)* represents type information deducible from the way that the value represented by *u* is used, so that we may use *backtype(u)* for each $u \in du(d,b)$ to determine the type of the output variable of *d*.

Next consider the case of a basic block *b1* with immediate successors *b1,...,bn* and a definition *d* which appears in *b1*. We again assume that *b1* does not appear in a program loop. We also assume that all uses to which *d* is chained appear in these n blocks. All the uses in *du(d,b1)* will contribute to our second method of type determination in deducing the type of *d* since each of them must use the value stored by *d*. However the uses in *du(d,b1),...,du(d,bn)* need not necessarily use the value stored by *d*. As already noted in section 3 (see Figure 3 and the associated discussion), this is because control may may flow from *b1* to any of *b2,...,bn* depending on the type of the value stored by *d*. The most that can be said about the type of the object created by *d* is that it is describable by the disjunction of the types indicated by the uses in each

of *b2,...,bn*. Under the above assumptions, we may formally state this assertion by defining:

(13) back(d) ≡ con([con: u∈du(d,b1)]backtype(u),
[dis:2≤i≤n]([con:u∈du(d,bi)]backtype(u)));

or, more generally, if *cesor(b)* represents the set of blocks which are immediate successors of *b* under the multivalued successor function, by defining:

(14) back(d) ≡ con([con:u∈du(d,b1)]backtype(u),
[dis:b∈cesor(b1)]([con:u∈du(d,b)]
backtype(u))).

As noted previously (in Section 3), if *b1* is a program exit; then none of the uses appearing in any of the blocks of *cesor(b1)* are valid determinants of the type of the output variable of *d*. Similarly, (see Figure 4 and the accompanying discussion), if a member of *cesor(b1)* appears on an alternate path from *b1* past a redefinition of the output variable of *d*, the uses which appear in that member of *cesor(b1)* are invalid in determining the type of the output variable of *d*.

If we drop our previous assumption that block *b* does not appear in a loop then one further possibility must be considered. Let us consider the case in which there does exist a path from *b* back to itself on which *d*

is the only definition which defines the output variable x of d. Let us further assume that uses of x occur in block b prior to the definition d. Such uses are clearly in $du(d,b)$ since they appear in b and may access the value stored by d. However it is possible that these uses do not access that value since control may flow from b along some alternative path which does not return to b before the output variable of d is redefined. Such uses act as if they occurred in a separate predecessor block of b and unlike those uses in $du(d,b)$ which appear after d, they cannot act as absolute determinants of the type of d but must have their deduced types disjoined with the deduced types of uses which exist along alternate paths. It is therefore convenient to introduce a formal means of "splitting" the block b into two separate blocks; one of which will contain all uses of the output variable of d which occur prior to d, and the other of which will contain all uses of the output variable of d occurring past d.

Given a program graph $G = \langle B, cesor \rangle$, where B is a set of basic blocks, $cesor$ is the successor function from B into 2^B and each block is an ordered set of definitions; and given a definition d in a block b $\in$ B, we may define G', *the graph of G divided at d* as follows:

Let $G' = \langle B', cesor' \rangle$ where $B' = (B - \{b\}) \cup \{b^+, b^-\}$ and $cesor'$ is a function from B' into $2^{B'}$ defined by:

(i) $cesor'(b') = cesor(b')$ if $b' \in B$ and $b \notin cesor(b')$;

(ii) $cesor'(b') = (cesor(b') - \{b\}) \cup \{b^-\}$ if $b' \in B$ and $b \in cesor(b')$

(iii) $cesor'(b^-) = \{b^+\}$;

(iv) $cesor'(b^+) = cesor(b)$ if $b \notin cesor(b)$; and

(v) $cesor'(b^+) = cesor(b) \cup \{b^-\}$ if $b \in cesor(b)$.

The block b^- includes all definitions in b occurring prior to and including d, and b^+ includes all definitions in b occuring past d. The essential property of the graph G' is that all uses chained to d which occur in a particular block may be treated uniformly in determining the type of d.

We define the function du' on the Cartesian product of the set of definitions by the set B' of all blocks in G' as follows:

$du'(d,b') = du(d,b')$ if $b' \in B$;

$du'(d,b^+) = \{u \in du(d,b) \mid u$ appears in $b^+\}$; and

$du'(d,b^-) = \{u \in du(d,b) \mid u$ appears in $b^-\}$.

Having completed these preliminaries, we can write a system of equations which describe the information concerning the type of a definition d at entries and exits of blocks of the graph divided at d. These equations are stated in terms of a function tfu_d defined on such block entries and exits. In what follows all references to blocks are to blocks of the program graph divided at

definition d. The equations which follow are straightforward generalizations of (12)-(14) and the accompanying discussions.

If b is a block, let $e(b)$ be its exit point and $y(b)$ its entry. The inverses of these two functions are indicated by $b(e)$ and $b(y)$ respectively. Then

(15) $$tfu_d(e) = tg$$

if e is a program exit or if $b(e)$ contains a redefinition of the output variable of d. This reflects the fact that no use occurring past a program exit or redefinition can contribute type information about a defined variable.

(16) $$tfu_d(e) = [dis:\ p \in cesor'(b(e))]tfu_d(y(p))$$

for all other block exits, e. This equation states that the type information known at the exit from a block is the disjunction of the information known at the entry points of all immediate successor blocks.

(17) $$tfu_d(y) = con(tfu_d(e(b(y))), [con: u \in du(d, b(y))] backtype(u))$$

for all block entries, y. This equation states that the type information known at entry to a block b is the conjunction of any information known at exit from b, together with information which can be deduced by examining uses within the block itself.

Finally, if b is the block of the original program

graph which contains a definition d, we define

(18) $$back(d) \equiv tfu_d(e(b^-)) = tfu_d(y(b^+)) \ .$$

The question of how to compute the function tfu_d algorithmically now arises. We postpone discussion of this problem until the next section and focus instead on the computation of $back(d)$. Note that $back(d)$ is defined by equations (15)-(18) in terms of the quantities $backtype(u)$ for the uses u of the program. $Backtype(u)$ is itself determined by the operator to which u is an input and by information concerning the type of the result produced by this operator. Thus, formulae (15)-(18) give a system of relationships among the types of the objects defined in a program. Moreover, $back(d_i)$ may be written as $g_i(x_1,\ldots,x_n)$ where, as before, x_i represents available information concerning the type of the output of d_i. Of course,

(19) $$type(d_i) \leq back(d_i)$$

since $back(d_i)$ only describes a range of type possibilities implied by the uses of the output variable of d_i and not the actual type of this variable.

Note also that, under the assumption of program correctness, information developed by the second method reduces the range of type possibilities which the first type determination method must confront. That is to say, inequalities (4) and (19) imply that

(20) $$type(d) \leq con(forward(d), back(d)) \ .$$

Let $h_i(x_1,\ldots,x_n) \equiv con(forward(d_i),back(d_i))$. Then our system of type determination relationship is expressed by the following inequalities:

(21) $$x_i \leq h_i(x_1,\ldots,x_n) \quad for \quad 1 \leq i \leq n \ .$$

To insure correctness of our type determinations we must choose a maximum solution to the system (21). By the corollary to Theorem 7 and a corollary to the proof of Theorem 6, if the functions h_i are monotonic, such a solution is given by:

(22) $$(x_1,\ldots,x_n) = (\hat{h}_1\ldots\hat{h}_n)^p(tg,\ldots,tg);$$

where p is the smallest integer such that

(23) $$x_i = h_i(x_1\ldots x_n) \quad for\ all \quad 1 \leq i \leq n \ .$$

Since the functions *con* and *forward* are both monotone in $x_1,\ldots,x_n$, the monotonicity of h_i will follow once we establish the monotonicity of *back*. By the definition of *back* ((15)-(18)), this will follow from the monotonicity of *backtype(u)* considered as a function of the type of the output variable of the definition in which u appears. Clearly, as the output variable of a definition is restricted in type, the input variable's types are restricted in corresponding fashion. This is true with one possible exception which occurs if the output variable is of type *tz*. In this case, the input variables may be

of arbitrary type, i.e. type *tg*. For example, *t1+ti = tz* will hold in our type calculus if *t1* is a set type, a tuple type, a bit string type or any other type except *ti* and *tg*. The disjunction of all these possible types is *tg*. However, if *t1 + ti = ti* holds, then *t1* must be of type *ti* or *tg* and *dis(ti,tg) = ti*. Since $ti \leq tg$, monotonicity does not hold in this case.

Recall, however, that the entire type determination algorithm is based on program correctness. Thus we must assume that an object is identified as being of type *tz* not because an error has occurred; but because the object has been left undefined. It is impossible for the output variable of a definition *d* to remain undefined unless that definition is never executed or if its inputs are undefined. If *d* cannot be executed, then there is no path from any definition to any of the uses *u* appearing in *d*, so that the definition of *backtype(u)* is immaterial. On the other hand, if *d*'s inputs were undefined then it is reasonable to define *backtype(u) = tz*. We therefore can adopt the convention that if the definition in which *u* appears is of type *tz*, then *backtype(u) = tz*. Since *tz* is the minimum element of our lattice, monotonicity of *backtype* holds.

We have therefore shown that (22) is indeed a solution to the system (21).

In general, however, (22) gives a very weak upper bound on the types of the definitions appearing in a program.

Indeed, if the types of all definitions are initially set to tg, the first method of type determination will not provide much information and the information developed by (22) is that of the second method alone. Further, we cannot effectively use the type of output variables to provide information about the types of the input variables since the types of all output variables are initialized to tg.

Fortunately, we can obtain a much sharper bound on the types of the definitions appearing in a program. An initial upper bound is obtained by using the first method alone. If we let $(y_1,\ldots,y_n) = (\hat{f}_1 \ldots \hat{f}_n)^k(tz,tz,\ldots,tz)$ as in (11), then to the system of inequalities in (21) we may add the system:

(24) $$x_i \leq y_i \quad for \quad 1 \leq i \leq n \ .$$

Note that

$$h_i(y_1,\ldots,y_n) = con(forward(d_i),back(d_i)) =$$

$$= con(f_i(y_1,\ldots,y_n),\ back(d_i)) \leq con(y_i,back(d_i)) \leq y_i \ ,$$

for all $1 \leq i \leq n$.

Thus, by a corollary to the proof of Theorem 6, the maximum solution to the combined system (21) and (24) is:

(25) $$(x_1,\ldots,x_n) = (\hat{h}_1 \ldots \hat{h}_n)^q(y_1,\ldots,y_n)$$

where q is the smallest integer such that

$$x_i = h_i(x_1,\ldots,x_n) \quad \text{for all} \quad 1 \leq i \leq n \ .$$

The vector $(x_1,\ldots,x_n)$ in (25) gives much more precise type information than the corresponding vector in (22).

8. Computing Types Algorithmically.

The general outline of an algorithmic process of type determination is now clear. Let $x_1,\ldots,x_n$ be type symbols representing the types of the definitions of the program $d_1,\ldots,d_n$. Let $f_i(x_1,\ldots,x_n)$ and $h_i(x_1,\ldots,x_n)$ be as defined in Sections 5 and 7 respectively. That is,

(26)
$$f_i(x_1,\ldots,x_n) \equiv forward(d_i) \ ,$$
$$h_i(x_1,\ldots,x_n) \equiv con(forward(d_i),back(d_i)) \ .$$

Then our algorithm may be given as follows:

Step 1: Repeatedly apply $(\hat{f}_1\ldots\hat{f}_n)$ to the vector of types which is initially $(tz,\ldots,tz)$ to obtain a vector $(y_1,\ldots,y_n) = (\hat{f}_1\ldots\hat{f}_n)(y_1,\ldots,y_n)$.

Step 2: Repeatedly apply $(\hat{h}_1\ldots\hat{h}_n)$ to the vector of types which is initially $(y_1,\ldots,y_n)$ to obtain the vector $(x_1,\ldots,x_n) = (\hat{h}_1\ldots\hat{h}_n)(x_1\ldots x_n)$. The type x_i is then an upper bound (in our lattice) of the set of types which the object defined by d_i may take on.

Note that if $y_i = tz$ for some $1 \leq i \leq n$ after step 1, an error will occur if d_i is ever executed. Of course, it may also be an indication that d_i cannot be executed because of the *cesor* function of the program graph. Therefore, if this condition is recognized after step 1 one of two situations confronts us. Either program correctness has been violated (so that there is no point in executing step 2 which uses the second method and therefore assumes program correctness); or definition d_i is inaccessible (so that it is pointless to attempt to apply the second method to it). Similarly, if in executing step 2, the type of some object is found to be tz, then that object has been used in ways which are incompatible; and we have detected an error for which a diagnostic can be emitted.

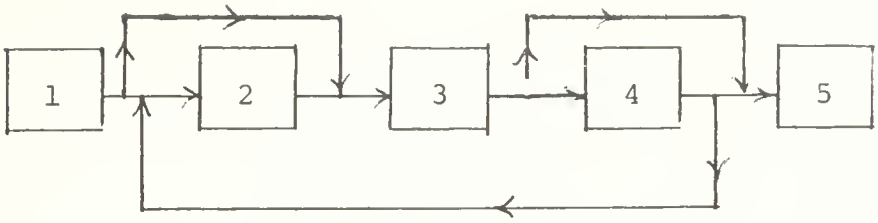
These observations allow us to sidestep the issue of the monotonicity of h_i which was raised at the end of Section 7. The only value for which the function h_i might fail to be monotonic is tz. However, except in the presence of a source error, h_i will never be applied to a vector which contains tz as a component. For greater elegance in the treatment of this type of error situation we may define h_i so that it is monotonic everywhere by defining *backtype(u)* $\equiv tz$ if the definition in which u appears is of type tz ; this was suggested towards the end of Section 7.

Several clarifications and refinements in the above general algorithm will now be made. The first concerns the method to be used for computing the function tfu_d defined by (15)-(17); this function is employed in defining $back(d_i)$ in (18) and thus ultimately in defining h_i.

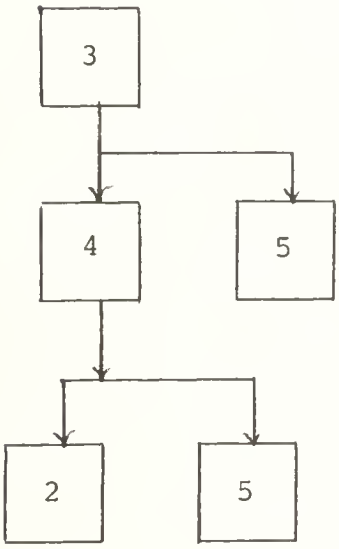
Let us assume that we have constructed the divided graph at a definition d. Our remaining task, therefore, is to solve equations (15)-(18) for that graph. The type information which may be deduced from a use u chained to d depends on what paths exist from d to u and what other uses exist along those paths and along any alternate paths from d. This information can be mechanically extracted from the divided graph by constructing the *tree of the program graph rooted at a node*. Given a program graph and a node of that graph, we form this tree in the following fashion. Establish the given node as the root of the tree and set the tree successors to be the graph successors as long as no cycle occurs. An example is shown in Figure 6.

To determine $back(d_i)$ we can form the graph divided at d_i, form the tree of the divided graph rooted at b^+, set tfu_d at the leaf exits to tg and apply equations (15)-(17) using the tree successor function and the entries and exits of tree nodes. $Back(d_i)$ is then the value of tfu_d at the entry of the root.

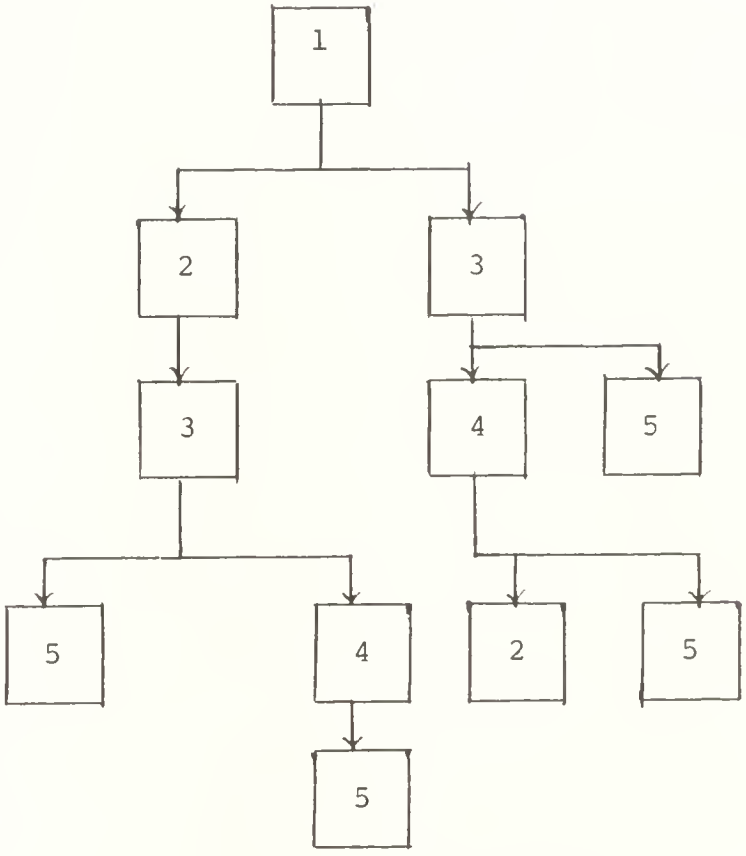
This method, however, still requires us to construct the divided graph for each definition d_i. The effect of



program graph



tree rooted at 3



tree rooted at 1

Figure 6. A Program Graph and its Program Trees Rooted at Nodes 3 and 1.

this division can be obtained in another fashion, which makes it unnecessary to actually build up the divided graph. Instead of creating the tree from a constructed divided graph, we create it from the original program graph. However, in "walking" the tree we treat the root node differently from any other node.

Recall that the original motivation for introducing the divided graph was to insure that all uses in a given block which are chained to a given definition are treated uniformly. In treating the set of uses $\{u\}$ chained to a definition d and which appear in the same block b as d we must separate {u} into two disjoint subsets. The uses u which appear in b past d are absolute determinants of the definition's type. This is because if d is executed these uses must subsequently be encountered and must use the value defined by d. However, uses u of d which appear in b prior to d need not necessarily use the output of d even if they are chained to d, This is because control may flow either from d to those uses or from some other definition to those uses, depending perhaps on the type of the defined variable.

We therefore compute separately the conjunctions of types indicated by those uses appearing in b past d and those uses appearing in b prior to d. The latter conjunction is ignored in the computation of tfu_d until we find some other node n which is an immediate graph predecessor

of b. The conjunction is then disjoined with the disjunction of types obtained from the tree successors of n. This process will be depicted more fully in the SETL type finding code presented in a later chapter.

Note, however, that we must still construct a separate tree for each node of the program graph. The use of graph theoretic spanning algorithms might greatly improve the efficiency of the above process but this possibility has not been investigated nor implemented. A simpler speedup, albeit one which loses some information, is to only construct the trees up to some maximum depth. This possibility has also not been investigated.

An additional refinement in the type analysis algorithm concerns the manner of application of $\hat{f}_1 \ldots \hat{f}_n$ and $\hat{h}_1 \ldots \hat{h}_n$. Each transformation $\hat{f}_i$ and $\hat{h}_i$ only changes one definition's type and involves only the application of f_i and h_i. Many of the types in the type status vector become fixed at their final values long before the entire system stabilizes. An extreme example is that of a value read into a variable and then output immediately, after which the variable is at once redefined; all within the same basic block. After a single iteration, the variable which is the target of the read operation is identified as being of type tg and it remains at that type throughout all succeeding iterations.

Let us assume that we have computed $(\hat{f}_1 \ldots \hat{f}_n)^j(tz,\ldots,tz)$ and wish to compute $(\hat{f}_1 \ldots \hat{f}_n)^{j+1}(tz,\ldots,tz)$. Let us further suppose that during the jth application of $(\hat{f}_1 \ldots \hat{f}_n)$, we have identified $\{\hat{f}_{i_1},\ldots,\hat{f}_{i_m}\}$ as being the only subset of $\{\hat{f}_1,\ldots,\hat{f}_n\}$ which could change the type status vector. Then, clearly, instead of applying $(\hat{f}_1 \ldots \hat{f}_n)$ for a $(j+1)$st time, we need only apply $(\hat{f}_{i_1} \ldots \hat{f}_{i_m})$. Our task, therefore, is to identify the subset $\{i_1 \ldots i_m\}$ of $\{1,,,.,n\}$.

For any definition df, let us define

$$s_1(df) \equiv \{d_i \in defs \mid \textit{one of the input uses to } d_i \textit{ is in } \bigcup_{b \in B} du(df,b)\}\ , \tag{27}$$

$$s_2(df) \equiv \{d_i \in defs \mid \textit{for some use } u \textit{ input to } df,\ d_i \in ud(u)\}.$$

A change of type of df represents a change of type to one of the input uses to every d_i in $s_1(df)$, so that $forward(d_1)$ and therefore f_i and h_i must be recomputed. Similarly, if u is an input use to df, $backtype(u)$ may have changed its value. If $d_i \in ud(u)$, this may cause a change in $back(d_i)$ so that h_i must be recomputed.

The overall algorithm may therefore be given as follows:

Step 1a: Initialize the types of all constants.
Initialize the types of all definitions to *tz*.
Set *workset* equal to the set of all definitions
in the program

1b: If *workset* = $\emptyset$ then go to step 2. Remove a definition *d* from *workset*. Set *oldtype* to the type of *d*. Set the type of *d* to *forward(d)*. If *oldtype* equals the type of *d* repeat step 1b.

1c: Set *workset* to *workset* $\cup$ $s_1(d)$. Repeat step 1b.

Step 2a: Set *workset* equal to the set of all definitions in the program.

2b: If *workset* = $\emptyset$ then halt.
Remove a definition *d* from *workset*.
Set *oldtype* to the type of *d*.
Set the type of *d* to *con(forward(d),back(d))*.
If *oldtype* equals the type of *d*, repeat step 2b.

2c: Set *workset* to *workset* $\cup$ $s_1(d)$ $\cup$ $s_2(d)$.
Repeat step 2b.

CHAPTER II

EXAMPLES

In this chapter we present a number of codes processed by the type determination algorithm of Chapter I, and describe the information concerning these codes which the algorithm was able to develop. This information was produced by a version of the algorithm written in the BALM language on the CDC 6600 at NYU. Each of the SETL routines to be presented was hand transcribed into quadruples (showing result operator and inputs) and the quadruples were manually grouped into basic blocks with a given successor function. The sets of basic blocks together with the successor function served as input to the BALM program. The output was a statement of the type of the output variable of each quadruple.

All the SETL routines processed were subroutines expecting one or more input parameters. Presumably, actual parameters would be passed by a calling routine. However, since our input did not include the calling routines it was necessary to insert preliminary quadruples which in effect performed a read operation for each of the parameters. The 'forward' method of type determination described in Chapter I can of course detect no information about an object which is read in. Moreover, because of the generality of SETL, the routine would work correctly on any one of

a large variety of different input object types. This means that if no indication of parameter types is given, we can expect only weak information to result from the way objects are used. For this reason we inserted quadruples which in effect "declared" the type of all parameters on entry to the routines. In an actual compiler, these types could either be determined by a type determination process to which the calling routine was available, or by a limited set of declarations governing subroutine parameters.

All secondary subroutines used in an example were inserted in-line.

We present our examples by first listing the SETL code analyzed in each case, and then noting what type information was found; types are described using the type symbols introduced in Chapter I. We also indicate the way in which the types were discovered and what aspects of the potentially available compile time type information were missed. In Section 7 we present a summary of all our results.

Example 1. Floyd's Treesort3.

Source: Algorithm 245, Collected Algorithms from CACM.

This algorithm sorts a tuple *m* of length *n*.

```
definef treesort(m,n);
     i = n/2;
     (while i ge 2 doing i = i-1;)
          siftup(i,n);;
     i = n;                                                    5
     (while i ge 2 doing i = i-1;)
          siftup(1,i);
          <m(1),m(i)> = <m(i),m(1)>;
     end while;
     return;                                                   10
end treesort;

definef siftup(i,n);
     copy = m(i);  ii = i;
   loop:  j = 2*ii;
     if j le n then                                            15
          if j lt n then
               if m(j+1) gt m(j) then j=j+1;;
          end if;
          if m(j) gt copy then m(ii) = m(j);
                               ii = j; go to loop; end if;     20
     end if;
     m(ii) = copy;
     return;
end siftup;
```

Line

1 *n* detected as *ti* from use in line 2

m detected as [*ti*]

2 *i* detected as *ti*

3 *i* ge 2 detected as *tb*; *i* detected as *ti*

5 *i* detected as *ti*

6 *i* ge 2 detected as *tb*; *i* detected as *ti*

8 *m(1)* and *m(i)* detected as *ti|tu* . Note that we did not detect that *i* lay in domain of the tuple *m*; *tu* is actually impossible. Similarly, after the assignment, *m* was detected as [*ti*]|*tt* rather than as [*ti*].

12 *i* and *n* detected as *ti*

13 *copy* detected as *ti|tu*; *ii* detected as *ti*

14 *j* detected as *ti*

15,16 *j* le *n* and *j* lt *n* detected as *tb*

17 *j*+1 detected as *ti*;

m(j+1) and *m(j)* detected as *ti|tu*;

m(j+1) gt *m(j)* detected as *tb* ; *j* detected as *ti*

19 *m(j)* detected as *ti|tu* ; *m(j)* gt *copy* as *tb* ;

m, after indexed assignment, as [*ti*]|*tt* .

20 *ii* detected as *ti*

22 *m* detected as [*ti*]|*tt* .

2. Huffman Encoding.

Source: On Programming,II, p. 149.

The algorithm is given a set of characters, *chars* and a frequency function *freq* which gives the number of times that each character appears in a particular text. The algorithm sets up an encoding *code* wherein those characters which appear most frequently have the shortest binary encoding.

```
definef huftables(chars,freq);
     work = chars;  wfreq = freq;  ℓ = nℓ;  r = nℓ;
     (while #work gt 1)
          c1 = getmin work;  c2 = getmin work;
          n = newat;  ℓ(n) = c1;  r(n) = c2;                    5
          wfreq(n) = wfreq(c1) + wfreq(c2);
          n in work;
     end while;
     code = nℓ;  seq = nulb;
     walk (∋ work is top);                                      10
     return <code,ℓ,r,top>;
end huftables;
```

```
definef getmin set;
     <keep,least> = <∋ set is x, wfreq(x)>;
     (∀x ∈ set)                                                    15
          if wfreq(x) lt least then <keep,least>=<x,wfreq(x)>;;
     end ∀;
     keep out set; return keep;
end getmin;

define  walk(top);  /* seq,l,r are global */                      20
     if ℓ(top) ne Ω then
          seq = seq + f;  walk(ℓ(top));
          seq = seq + t;  walk(r(top));
     else code(top) = seq;
     end if;                                                       25
     seq = seq(1: #seq-1);
     return;
end walk;
```

Line

1 *chars* declared as {*tc*}; *freq* as {<*tc*,*ti*>}

2 *work* detected as {*tc*}; *wfreq* as {<*tc*,*ti*>}

l as *tn*; r as *tn*;

3 *#work* detected as *ti*; logical as *tb*;

4 *c1* and *c2* detected as *tc*|*ta*|*tu*

5 *n* detected as *ta*; *l* and *r* as {<*ta*,*ta*|*tc*>}|*tn*

6 *wfreq(c1)* and *wfreq(c2)* detected as *ti*;

wfreq detected as {<*ta*|*tc*,*ti*>}

7 *work* detected as {*ta*|*tc*}

9 *code* detected as *tn*; *seq* as *tb*

10 *top* detected as *ta*|*tc*

11 <*code*,*l*,*r*,*top*> detected as <{<*ta*|*tc*,*tb*>}|*tn*,

{<*ta*,*ta*|*tc*>}|*tn*, {<*ta*,*ta*|*tc*>}|*tn*, *ta*|*tc*>

14 *x*, *keep* detected as *ta*|*tc*|*tu*; *least* as *ti*|*tu*

16 logical recognized as *tb*; first *wfreq* as *ti*

second as *ti*|*tu* ; *keep* as *ta*|*tc*|*tu* ; *least* as *ti*|*tu*

18 *set* detected as {*ta*|*tc*}|*tn*

21 *l(top)* detected as *ta*|*tc*|*tu*; *logical* as *tb*

22 *seq* detected as *tb*; *l(top)* as *ta*|*tc*|*tu*

23 *seq* detected as *tb*; *r(top)* as *ta*|*tc*|*tu*

24 *code* detected as {<*ta*|*tc*,*tb*>}

26 *seq* detected as *tb*

In converting this example into quadruples, the stacks for implementing recursion were explicitly programmed. The stack for the variable *top* was found to be [*ta*|*tc*]|*tt*.

3. Multiplication of Permutations.

Source: On Programming, II, p. 146.

The algorithm is given an ordered tuple, *seqperms*. Each component of this tuple is itself a set representing a permutation in cyclic form. The elements of such a set are tuples representing disjoint cycles. The algorithm multiplies the permutations and returns the resultant permutation in cyclic form.

```
definef multall(seqperms);
     seq = nult;  marked = nl;
     (∀perm(m) ∈ seqperms, cyc ∈ perm)
          (∀elt(m) ∈ cyc) seq(#seq+1) = elt;;
          seq(#seq+1) = cyc(1);                                   5
          (#seq) in marked;
     end ∀;
     result = nl;
     (while ∃e(m) ∈ seq | n m ∈ marked)
          cyc = <seq(m)>;  elt = seq(m+1);                        10
          loc = m+1;  m in marked;
     <loop:>(loc < ∀n < #seq|seq(n) eq elt and n n∈marked)
               n in marked;
               elt = seq(n+1);
          end ∀;                                                  15
       if elt ne cyc(1) then cyc(#cyc+1) = elt; loc=1; go to loop;
          else cyc in result;;
     end while;
     return result;
end multall;                                                      20
```

Line

1 *seqperms* declared as [{[*ti*]}]

2 *seq* detected as *tt*; *marked* as *tn*

3 *m* detected as *ti*; *perm* as {[*ti*]}|*tu* ;
cyc as [*ti*]|*tu*. Note in line 3 we index a tuple by an integer in its domain and take an arbitrary member of a set in a ∀ loop header. These operations cannot give rise to Ω, but this fact is not detected by our algorithm.

4 *m* and #*seq*+1 detected as *ti*; *elt* as *ti*|*tu*.

5 *seq* detected as [ti]|tt ; *cyc(1)* as *ti*|*tu*

6 *marked* detected as {*ti*}

8 *result* detected as *tn*

9 *m* detected as *ti*; *e* as *ti*|*tu* ; logical as *tb*.

10 *cyc* detected as <*ti*|*tu*> ; *elt* as *ti*|*tu*

11 *loc* detected as *ti* ; *marked* as {*ti*}

12 *n* detected as *ti* ; logicals as *tb*

13 *marked* detected as {*ti*}

14 *elt* detected as *ti*|*tu*

16 *cyc(1)* detected as *ti*|*tu*; *loc* as *ti*;
cyc as [*ti*|*tu*]|*tt*

17 *result* detected as {[*ti*|*tu*]|*tt*}

4. Interval Analysis

Source: On Programming, II, pp. 269-271.

This algorithm is given a program graph in the form of a set of *nodes*, an *entry* node and a graph successor function, *cesor*. It returns the set of intervals of the graph. Each such interval is represented as a tuple of nodes.

```
definef intervals(nodes,entry,cesor);
     ints = nl; seen = {entry};  follow = nl;  intov = nl;
     (while seen ne nl) node from seen;
          interval(nodes,node, cesor) is i in ints;
          follow(i) = followers;                              5
          (1 ≤ ∀k ≤ #i) intov(i(k)) = i;;
          seen = seen + followers;
     end while;
     return ints;
end intervals;                                                10

definef interval(nodes,x,cesor);
     npreds = {<x,0>, x ∈∈nodes};
     (∀x ∈ nodes, y ∈ cesor(x)) npreds(y) = npreds(y)+1;;
     int = nult; followers = {x}; count = {<y,0>, y ∈ nodes};
     count(x) = npreds(x);                                    15
     (while {y∈followers|npredes(y) eq count(y)}is newin ne nl)
          (∀z∈newin) int(#int+1) = z;  z out followers;
               (∀y∈cesor(z)|y ne x)count(y)=count(y)+1;
                                        y in followers;
```

```
        end ∀z;                                                    20
    end while;
    return int;
end interval;
```

line

1 *nodes* declared as {*ti*}; *entry* as *ti*;
cesor as {<*ti*,{*ti*}|*tn*>}

2 *ints*, *follow*, *intov* detected as *tn*; *seen* as {*ti*}

3 logical detected as *tb* ; *node* as *ti*|*tu* . Note that although *seen* cannot be the empty set at this point, so that *node* should really be *ti* , our algorithm cannot detect this.

4 *i* detected as [*ti*]|*tt*; *ints* as {[*ti*]|*tt*}

5 *follow* detected as {<[*ti*]|*tt*>, {*ti*}|*tn*}

6 *k* detected as *ti* ; *i(k)* as *ti*|*tu* ;
intov as {<*ti*,[*ti*]|*tt*>}.

7 *seen* detected as {*ti*}|*tn*.

12 *npreds* detected as {<*ti*|*tu*,*ti*>}

13 *x*,*y* detected as *ti*|*tu* ; *npreds* as {<*ti*|*tu*,*ti*>}

14 *int* detected as *tt* ; *followers* as {*ti*} ;
count as {<*ti*|*tu*,*ti*>}

15 *count* detected as {<*ti*|*tu*,*ti*>}|*tn*; *npreds* as *ti*|*tu*

16 logicals detected as *tb*; *newin* as {*ti*}|*tn*

17 *int* detected as [*ti*]|*tt*; *z* as *ti*|*tu*;
followers as {*ti*}|*tn*

18 logical detected as *tb* ; *y* as *ti*|*tu* ;
count as {<*ti*|*tu*,*ti*>} ; *followers* as {*ti*}.

5. Ford-Johnson Tournament Sort

Source: On Programming, II, p. 67.

The program is given an integer sequence *items* and returns a sorted sequence.

```
definef fordj(items);
    if (#items) eq 1 then return items;;
    au = nl;  bu = nl;  i = 1;
    (while i lt #items doing i = i+2;)
        x = items(i);  y = items(i+1);
        if x lt y then <x,y> = <y,x>;;
        au(#au+1) = x;   bu(#bu+1) = y;
    end while;
    oddone = items(i);  a = fordj(au);  b = nl;
    (1 ≤ ∀j ≤ #a)
        dummy = 1 ≤ ∃n ≤ #a|au(n) eq a(j);
        au(n) = Ω;        b(n) = bu(j);
    end ∀j;
    b(#b+1) = oddone;
    lia = 1;  jbot = 1;  jtop = 1;  length = 1;
    (while jbot le #b)
        (jtop ≥ ∀j ≥ jbot) low = lia-1; high = j;
            (while (high-low) gt 1) mid=(high+low)/2;
                if b(j) le a(mid) then high = mid;
                    else low = mid;;
            end while;
```

```
                (lia ≤ ∀i ≤ low)a(i-1) = a(i);;
                a(low) = b(j);  lia = lia-1;
            end ∀j;
            jbot = jtop+1;  length = 2*length + 1;                    25
            jtop = (lia + length) min #b;
        end while;
        return {<p(1)-lia+1, p(2)>, p ∈ a};
    end fordj;
```

Line

1 *items* declared as {<*ti*,*ti*>}

2 #*items* detected as *ti*; logical as *tb*

3 *au*,*bu* detected as *tn* ; *i* as *ti*

4 logical detected as *tb*; *i* as *ti*

5 *x*,*y* detected as *ti*

6 logical detected as *tb* ; *x*,*y* as *ti*

8 *au* and *bu* detected as {<*ti*,*ti*>}

9 *oddone* detected as *ti*|*tu*; *a* as {<*ti*,*ti*|*tu*>}|*tn*

10 *j* detected as *ti*

11 *dummy* detected as *tb* ; *n* as *ti*

12 *au*,*b* detected as {<*ti*,*ti*>}|*tn* ; *buj* as *ti*|*tu*

14 *b* detected as {<*ti*,*ti*>}|*tn*

15 *lia*, *jbot*, *jtop*, *length* detected as *ti*

16 logical detected as *tb* ; #*b* as *ti*

17 *j* detected as *ti*; *low*,*high* as *ti*

18 logical detected as *tb* ; *mid* as *ti*

19 logical recognized as *tb*; *b(j)*,*a(mid)*,*high*,*low* as *ti*

22 *i* detected as *ti* ; *a(i)* as *ti|tu* ; *a* as {<*ti*,*ti|tu*>}|*tn*

23 *lia* detected as *ti*; *b(j)* as *ti|tu* ; *a* as {<*ti*,*ti|tu*>}|*tn*

25 *jbot*, *length* detected as *ti*

26 *jtop* detected as *ti*

28 *p(1)* detected as *ti* ; *p(2)* as *ti|tu* ; set as {<*ti*,*ti|tu*>}

6. Topological Connectivity Analysis

Source: On Programming, II, pp. 262-263.

This algorithm is given a mapping *color* from a set of bricks, each represented by an ordered pair giving its position, into the Boolean set {t,f} where t represents the color black and f, the color white. *nrows* and *ncols* are the number of rows and columns respectively. The routine returns a mapping such that all the bricks in a connected region of a single color are mapped onto a single representative brick of the region.

```
definef topanalyze(color,nrows,ncols);
     inside = nl; represent = nl;
     (1 ≤ ∀nc ≤ ncols) represent(<1,nc>) = <1,1>;;
     poffset = 0;
     (2 ≤ ∀nr ≤ nrows) represent(<nr,1>) = <1,1>;          5
          (2 ≤ ∀nr ≤ ncols)
               pnaybs=<<nr-1,nc-poffset>,
                 <nr-1,nc-poffset+1 min ncols>,<nr,nc-1>>;
```

```
            if n  nayb(n)∈pnaybs|color(nayb) eq
                        color(<nr,nc>) then continue;
              else represent(<nr,nc>) = nayb;
                if n eq 2 and color(pnayb(3)) eq
                                    color(pnayb(2))
                  then if truerep(nayb) ne truerep(pnayb(3))
                          then represent(pnayb(3))=nayb;
                          else <nayb,pnayb(1)>in inside;
                end if; end if;
            end if;
        end ∀nc;
        poffset = 1-poffset;
    end ∀nr;
    return {<truerep(x(1)),truerep(x(2))>, x ∈ inside};
end topanalyze;

definef truerep(brick);
    b = brick;
    (while represent(b) ne Ω) b = represent(b);;
    return b;
end truerep;
```

Line

1 *color* declared as {<<*ti*,*ti*>, *tb*>}; *nrows*,*ncols* as *ti*

2 *inside*, *represent* detected as *tn*

3 *nc* detected as *ti* ; *represent* as {<<*ti*,*ti*>,<*ti*,*ti*>>}

4 *poffset* detected as *ti*

5 *nr* detected as *ti* ; *represent* {<<*ti*,*ti*>,<*ti*,*ti*>>}

6 *nc* detected as *ti*

7 *pnaybs* detected as <<*ti*,*ti*>,<*ti*,*ti*>,<*ti*,*ti*>>

9 logicals detected as *tb*; *n* detected as *ti* ;
nayb as <*ti*,*ti*>|*tu*; *color(nayb)* and *color(<nr,nc>)*
as *tb*|*tu*

11 *represent* detected as {<<*ti*,*ti*>,<*ti*,*ti*>>}|*tn*

12 logicals recognized as *tb* ; *pnayb(2)*, *pnayb(3)*
as <*ti*,*ti*>|*tu*

14 logicals detected as *tb*

15 *represent* detected as {<<*ti*,*ti*>,<*ti*,*ti*>>}|*tn*

16 *inside* detected as {<<*ti*,*ti*>|*tu*,<*ti*,*ti*>|*tu*>}

20 *poffset* detected as *ti*

22 set detected as {<<*ti*,*ti*>|*tu*,<*ti*,*ti*>|*tu*>}

25,26 *b* recognized as <*ti*,*ti*>|*tu* ; logical recognized as *tb*

7. Summary of the Results.

In this section we present some statistics and summarize the results obtained using the typefinder on the examples of the previous 6 sections. In general, we may say that, when given only a small amount of declaratory information (in most cases, a declaration of the types of the parameters of a subroutine), the typefinder performs rather well.

Of course the type information assigned to a variable is always 'larger' (in the sense of the type lattice) than the disjunction of the actual types which the variable's value can take on during program execution. In over 70% of the cases, the detected types were "exact fits" -- that is, they were the best description possible within our type calculus for the actual run time types of objects.

There were three distinct groups of cases in which the best description was not obtained.

1) An object defined by a function application is often described as possibly being of type *tu* (the undefined atom) even when a more precise static analysis would show that the function parameter will be within the domain of the function. A good example of this occurs in line 6 of section 4:

```
(1 ≤ ∀k ≤ #i) intov(i(k)) = i;;
```

Here, *i(k)* is described as being of type *ti|tu* despite the fact that *i* is known to be of type *[ti]|tt* (line 4) and *k* is clearly in range. Similar situations occur in

Section 1, lines 8, 13, 17; Section 2, lines 16, 22, 23; Section 3, lines 3, 4, 9, 10; Section 4, lines 3, 13, etc.

In many cases the fact that such an object is not of type *tu* could be established by examining its subsequent uses. An example of this occurs in Section 2, line 16:

```
if wfreq(x) lt least then <keep,least> = <x,wfreq(x)>;;
```

Here, the first occurrence of *wfreq(x)* is recognized as being of type *ti* because it is used in a comparison operation; but the second occurrence is described as being of type *ti|tu*.

Another very common instance of this type of imprecision occurs in 'forall' loops. Consider again Section 2, lines 15-17:

```
(∀x ∈ set)
     if wfreq(x) lt least then <keep,least> =<x,wfreq(x)>;;
end ∀;
```

which when expanded for analysis reads as follows:

```
1    temp = set;
2  loop: if temp ne nℓ then x = arb temp;
3        temp = temp less x; if wfreq(x) lt least etc.;;
4        go to loop;
5    end if;
```

If the instruction x = arb *temp* in line 2 is executed, *temp* cannot be equal to nℓ and thus x cannot equal Ω. However this is not detected by our typefinder, since the fact that *temp* ≠ nℓ depends on the semantics of the test. Thus we

find that in Section 2, line 16, *keep* is described as *ta|tc|tu* rather than *ta|tc*.

We will call this kind of imprecision, in which an object is incorrectly described as possibly being of type *tu*, a *type I imprecision*. Of course, it is not always the case that such an identification is incorrect. For example, consider Section 2, line 21 where *l(top)* is correctly identified as being of type *ta|tc|tu* , although the identification of *l(top)* in line 22 as *ta|tc|tu* represents a type I imprecision. Similarly in line 9 of Section 5, *oddone* is correctly identified as being of type *ti|tu*.

2) A *type II imprecision* occurs when a set or a tuple is incorrectly identified as possibly being null. Often this is a direct result of a type I imprecision, as in the following example, taken from Section 4, lines 16-17:

```
16   (while {y∈followers|npreds(y) eq count(y)} is newin ne nℓ)
17         (∀z∈newin) int(#int+1) = z;  z out followers;
```

In line 17, *newin* cannot be of type *tn* (the null set) but this fact is not detected, so that *z* is identified as being of type *ti|tu* (a type I imprecision). When the indexed assignment is then made to *int*, *int* should be identifiable as [*ti*]. However, since our analysis algorithm judges that we might be assigning Ω to the first position of a tuple of length 1, *int* is incorrectly identified as of type [*ti*]|*tt*.

Other examples of type II imprecisions are in Section 1, lines 8, 19, 22; Section 2, lines 5, 18; Section 3, line 16; Section 5, lines 9, 12, 22, 23, etc.

3. A third type of imprecision occurs when an object which is incorrectly identified by a type I or type II imprecision is inserted into a set or tuple. An example occurs in Section 5, line 28:

```
return {<p(1)-1ia+1,p(z)>, p ∈ a};
```

p(1) is detected as being of type *ti* from its use in a subtraction operation. *p(2)*, however, is detected as of type *ti|tu* -- a type I imprecision. The set which is returned is therefore incorrectly described as being of type {<*ti*,*ti|tu*>}. (Note that in our type calculus we do not convert <*ti*,*ti|tu*> to [*ti*] since the former gives more information than the latter.)

Such imprecisions will be disregarded in the statistics which follow, since the objects which they identify are correctly determined as to their primary type -- non-null set or non-null tuple.

For each of our six examples the following table indicates the number of quadruples, the number of exact detections, the number of type I and type II imprecisions and the percentage of exact detections.

	Section						
	1	2	3	4	5	6	Total
Number of quads	45	62	54	74	92	76	403
# of exact detections	33	47	43	56	76	55	310
# of type I imprec.	6	10	8	14	7	18	63
# of type II imprec.	6	5	3	4	9	3	30
% of exact detections	73%	76%	82%	73%	83%	72%	77%

In Chapter V we will present several proposals for reducing the number of imprecisions.

CHAPTER III

A SETL SPECIFICATION OF THE TYPE-FINDING ALGORITHM

1. Representations of Programs and Preliminary Processing.

In this chapter we specify the type determination algorithm in SETL. A running SETLB version of this specification exists and was used in the development of the 'production' variant of the algorithm. We shall give our specification in a top-down style, first presenting the global algorithm and then specifying the various necessary subroutines.

A program is represented as a 5-tuple of the form *<nodes,progrph,entry,cesor,cons,exits>*, where:

nodes is the set of basic blocks of the program;

progrph is a function which maps each member of *nodes* into a tuple representing the operations occurring in that basic block in order of execution. Such a tuple will be called a *block tuple*. Each operation is itself represented by a tuple consisting of an output variable, an operator, and input variables. Such an operation will be called a *defining tuple*;

entry is an element of *nodes* which is the entry block to the program;

cesor is a mapping from *nodes* into 2^{nodes} which for each basic block, gives the set of its successors;

cons is the set of constants occurring in the program;

exits is the set of exit blocks of the program.

Preliminary processing of the program 5-tuple, by a method incorporating interval analysis and use-definition chaining, produces the following auxiliary objects.

defs the set of all definitions appearing in the program. A *definition* corresponds to a defining tuple, but is represented by a triple consisting of a defined variable, the basic block in which the defining tuple occurs and an integer which gives the position of the defining tuple within the basic block. A *use* of a variable is similarly represented by a quadruple consisting of the variable used, the basic block within which the use appears, an integer which gives the position among all the defining tuples of a basic block of the defining tuple t within which the use appears, and an integer which gives the position of the use among all the uses which are part of t. For example if *prgrph(b)(4)* = <y,op,z,w,t> then the definition of y is represented by <y,b,4> and the use of w by <w,b,4,2>. To simplify the code which follows we shall make use of the following macros:

```
+*blk(x) = x(2) ** /* the block in which x appears */
+*blkpos(x) = x(3)**/* the  position of x within the block*/
+*usepos(x) = x(4)** /* the position of a use x among the
                        uses of a definition */
```

ud a mapping which associates with each use of a variable the set of all definitions which are chained to that use (see Chapter I, Section 4).

du a mapping which associates with each definition of a variable the set of all uses which are chained to that definition.

constyp a mapping which, given a SETL constant, produces its type using the type representation which will be discussed below.

The algorithms for obtaining these auxiliary items are straightforward extensions of the algorithms presented in (Schwartz, 1973b).

The SETL operators with which we deal in the running SETLB typefinder are represented by integers. We name these and describe their meanings as follows:

ohd	SETL hd operation	*oarb*	SETL arb operation
otl	SETL tl operation	*oass*	assignment operation
oad	plus operator	*osb*	minus operator
oml	asterisk operator	*odv*	slash operator
orm	double slash operator	*opw*	SETL pow operation
onpw	SETL npow operation	*oset*	SETL set former
otpl	SETL tuple former	*omxm*	SETL max operation
omnm	SETL min operation	*oabs*	SETL abs operation

osiz	# operator	*odec*	SETL dec operation
ooct	SETL oct operation	*onot*	SETL not operation
oor	SETL or operation	*oand*	SETL and operation
oelm	∈ operator	*ord*	SETL read operation
oeq	comparison operators	*owth*	SETL with operation
one		*olss*	SETL less operation
olt		*olsf*	SETL lesf operation
ole		*oinc*	SETL incs operation
ogt		*oof*	functional application e.g. f(x)
oge			
ondx	substring/subtuple op	*oofa*	multivalued functional application, e.g. f{x}.
odxs	indexed assignment		

These operators can be categorized according to the number of their operands. *ord* has no arguments so that read x is represented by the defining tuple $\langle x, ord \rangle$. Simple assignments such as $x = y$ are represented by $\langle x, oass, y \rangle$. Unary operators are combined with assignment so that $x =$ hd y is represented by $\langle x, ohd, y \rangle$. The operators which take only one argument are *ohd*, *otl*, *oarb*, *oass*, *opw*, *odec*, *ooct*, *oabs*, *osiz* and *onot*. There is a single ternary operator, *ondx*; $x = y(z{:}w)$ is represented by $\langle x, ondx, y, z, w \rangle$. Three other operators have an indefinite number of arguments; these are *oset*, *otpl*, and *odxs*. $x = \{y, z, w, t\}$ is represented by $\langle x, oset, y, z, w, t \rangle$ and similarly for $x = \langle y, z, w, t \rangle$. Indexed assignment (*odxs*)

is unusual in that it is the only operator in which the previous type of the defined variable is a determinant of its type after the operation is performed. E.g. if x is a tuple, $x(3) = y$ causes x to remain a tuple, and if x is a set then the operation causes x to remain a set. For this reason, the defined variable is repeated as one of the input uses in the defining tuple. For example, $f(x) = z$ is represented by $<f,odxs,x,z,f>$ and $f(x,y) = z$ by $<f,odxs,x,y,z,f>$. All other operators are binary.

The details of the operators used and the manner in which defining tuples are represented are of course unimportant in discussing the high level logic of the type determination algorithm; but these details are critical in actually programming and implementing such an algorithm.

2. Representation of Type Symbols.

We now give a scheme for representing type symbols as we have defined them in Chapter 1, Section 2. The scheme has the advantage of enabling us to perform alternations elegantly and efficiently. Other schemes might do just as well, but the *con* and *dis* routines below would have to accommodate a particular method of representation.

Elementary types except for *tz* and *tg* are represented by distinct bit strings of length 8 (which is the number

of elementary type symbols other than *tz*). A bit string representing an elementary type symbol has exactly one nonzero bit. Thus *ti* might be represented by bit string 00000010b while *tu* might be represented by 00000001b. *tz* is represented by a bit string of all zeros and *tg* by a bit string of all ones. The alternation of two elementary types is represented by the inclusive "or" of the two representations. This scheme automatically ensures that *tg* and *tz* are the absorptive and identity elements for alternation.

We associate the names of type symbols with their bit string representations allowing us to refer to the objects *tu*, *ti*, *tn*, *tt*, *tg*, *ta*, *tz*, *tb* and *tc* in the ensuing algorithms.

If *t* is a type symbol and *r* its SETL representation, we represent the type symbol {*t*} by the SETL object <2, *tz*, *r*> and the type symbol [*t*] by the SETL object <3, *tz*, *r*>. If *t1*,...,*tn* are type symbols with representations *r1*,...,*rn* respectively then the type symbol <*t1*,...,*tn*> will be represented by the SETL object <4, *tz*, *r1*,...,*rn*>. An alternation between a grosstype represented by <*r(1)*,*tz*,*r(3)*,...> and an elementary type represented by *rr* will be represented by the SETL object <*r(1)*,*rr*,*r(3)*,...>. For example, <2,*tn*,*ti*> represents the type symbol {*ti*}|*tn* and <4,*tt*,*ti* or *tb*,*ti*> represents the type symbol <*ti*|*tb*,*ti*>|*tt* .

We define some useful macros:

```
=* set = 2 **      +* unt = 3 **      +* knt = 4 **
=* isel(x) = type x eq bits **
+* isset(x) = if isel(x) then f else x(1) eq 2 **
+* isunt(x) = if isel(x) then f else x(1) eq 3 **
+* isknt(x) = if isel(x) then f else x(1) eq 4 **
+* grostyp(x) = if isel(x) then 1 else x(1) **
```

We now present the routines for disjunction and conjunction. First we give a SETL routine *dis* which when given two type representations returns the representation of their disjunction (see Chapter 1, Section 2).

```
definef dis(a,b);
     if  grostyp(a) gt grostyp(b) then return dis(b,a);;
     if   el(a) then /* form the alternation of a and b */
          if isel(b) then return a or b;;
          return <b(1), a or b(2)> + tl tl  b;;
     if (isset(a) and isset(b)) or (isunt(a) and isunt(b)) then
          /* disjunction of two sets or of two tuples
               of unknown length */
          return <a(1),a(2) or b(2), dis(a(3),b(3))>;;
     if isunt(a) then /* disjunction of an unknown and known
               length tuple */
          return <unt, a(2) or b(2),dis(a(3),
                          [dis: 2<i≤#b]b(i))>;;
```

```
    if isknt(a) then return /* a and b are both known
                                length tuples */
        if #a eq #b then /* disjunction of two tuples
                            of same length */
          <knt,a(2) or b(2)>+[+: 2<i≤#a]<dis(a(i),b(i))>
        else /* disjunction of two tuples of differing
                                                lengths */
          <unt,a(2) or b(2), dis([dis: 2<i≤#a]a(i),
                              [dis: 2<i≤#b]b(i))>;;
    /* otherwise the disjunction reduces to the general type*/
    return tg;
end dis;
```

We also make *dis* available as an infix operator.

```
define a dis b;
    return dis(a,b);
end;
```

Next we present the SETL routine *con*, which when given two type symbol representations returns the representation of their conjunction (see Chpater 1, Section 2).

```
definef con(a,b);
    if grostyp(a) gt grostyp(b) then return con(b,a);;
    if isel(a) and isel(b) then return a and b;;
    ela = if isel(a) then a else a(2); /* elementary
                                alternand of a */
    /* at this point b is known to be non-elementary */
```

```
    eltyp = /* the elementary alternand of the result */
         ela and b(2);
    if isknt(b) then tup = <knt,eltyp>;
         if isunt(a) then
         /* conjunction of unknown and known length tuple */
           (2 < ∀i ≤ #b) tup(i) = con(b(i),a(3));;;
         if isknt(a) and #a eq #b then /* two tuples of same
                                                  length */
           (2 < ∀i ≤ #a) tup(i) = con(a(i),b(i));;;
         if #tup eq 2 or 2 < ∃i ≤ #tup| tup(i) eq tz
              then return eltyp;
              else return tup;;
    end if;
    if grostyp(a) eq grostyp(b) then
         if con(a(3),b(3)) eq tz then return eltyp;
              else return <a(1),eltyp,con(a(3),b(3))>;;
    end if;
    return eltyp;
end con;
```

We also make *con* available as an infix operator;

```
define a con b;
    return con(a,b);
end;
```

3. The Global Type Determination Algorithm.

Before presenting SETL code for the type determination algorithm, we describe the mapping *typ* which will be constructed by that algorithm. This mapping from the set *defs + cons* into the set of type representations corresponds to the 'type status vector' of Chapter 1, Definition 11. If x is a definition or a constant, *typ(x)* is the type of that object as determined by our algorithm at a given point. Recall that the mapping *constyp* when given an element of *cons* returns the type representation of that constant.

The type determination algorithm is as described in Chapter 1, Section 8, but is preceded by a short initialization section. The steps noted in the comments below are the steps of that algorithm as presented at the end of that section, to which the reader is referred. In steps 1c and 2c we use two sets, $S_1(df)$ and $S_2(df)$ which were defined by

$$s_1(df) \equiv \{d_i \in defs \mid \text{one of the input uses to } d_i \text{ is in } du(df)\}$$

and

$$s_2(df) \equiv \{d_i \in defs \mid \text{for some use } u \text{ input to } df,\ d_i \in ud(u)\}.$$

We now give SETL code to compute these two sets, and then give the code for the type determination algorithm itself. The variables *defs*, *ud*, *du*, *progrph* and *constyp* are assumed global. We use an auxiliary macro defined by

+*deftup(x) = prgrph(blk(x))(blkpos(x))** which, when given a definition or use x, returns either the defining tuple which x represents (if x is a definition), or (if x is a use) returns the defining tuple in which x occurs.

```
definef sl(df);
     /* returns the set of definitions which may use the
          quantity defined by df */
     return [+: u ∈ du(df)]{d∈defs|tℓ df eq u(2:2)};
end sl;

definef s2(df); /* returns the set of definitions chained
               to the uses which are inputs to df */
     used = /* the tuple of variables which are inputs to d */
            tℓ tℓ deftup(df);
     return [+: v(j)∈used] ud(<v, blk(df), blkpos(df),j>);
end s2;
```

The type determination algorithm uses routines *forward(d)* and *back(d)* to compute the type of d as determined by the first and second methods of type determination respectively (see Chapter I, Sections 5 and 7). These routines will be given shortly.

```
define  typedeterm;
     /* initialization section */
     typ = nl;
     (∀x ∈ cons) typ(x) = constyp(x);; /* initialize types
                              of constants */
     (∀x ∈ defs) typ(x) = tz;; /* initialize type status
                              to <tz,...,tz> */
     /* end of initialization */
     work = defs;  /* step 1a */
     (while work ne nl)
          /* step 1b */
          d from work;
          oldtype = typ(d);
          typ(d) = forward(d);
          if oldtype ne typ(d) then /* step 1c */
               /* adjust work to include those definitions
                    whose type may be changed in the
                    the next forward pass */
               work = work + sl(d);
          end if;
     end while;
     /* we now begin step 2, the joint forward and backward pass*/
     work =defs;  /* step 2a */
     (while work ne nl)
          /* step 2b */
          d from work;
```

```
            oldtype = typ(d);
            typ(d) = forward(d);
            if oldtype ne typ(d) then /* step 2c */
                 /* adjust work to include those definitions
                      whose type may be changed in the next
                      forward or backward pass */
                 work = work + s1(d) + s2(d);
            end if;
       end while;
end typedeterm;
```

4. The First Method of Type Determination.

In specifying the routine *forward(d)* which gives the type of definition *d* as determined by the first method of type determination, we make use of a function *calc* as specified in Chapter I, Section 5. We assume the existence of a SETL routine *calc* which takes two arguments. The first, *op* is an operator representation; and the second, *types*, is a tuple of type representations which are the types of the variables input to *op*. *calc* returns that type representation which designates an object resulting from operator *op* acting on objects of type *types*.

The *calc* function is of course dependent on the programming language the types of whose variables we are determining. The body of the function consists essentially

of a collection of tables enumerating all basic combinations of type and operator. Because of its length and detailed nature, *calc* will be given in an appendix.

The routine *forward* notes the operator and input of a definition *d*, determines the types of the inputs using formula (3) of Chapter 1 and calls *calc* in accordance with formula (4) of Chapter 1. *ud* and *progrph* are assumed global.

```
definef forward(d);
     types = nult;  /* initialization */
     tup = /* the defining tuple corresponding to d */
          deftup(d);
     uses = /* the tuple of input variables */
          tℓ tℓ tup;
     (∀v(j) ∈ uses)
          u = /* the use which the jth variable of tup represents*/
              <v,blk(d),blkpos(d),j>;
          /* apply formula (3) of Chapter 1 */
          types(j) = [dis: df ∈ ud(u)] typ(df);
     end ∀;
     op = /* the operator of d */
          tup(2);
     /* we now call calc in accordance with formula (4) */
     return calc(op,types);
end forward;
```

5. The Second Method of Type Determination

We now give SETL code for the second method of type determination which was presented in Chapter 1, Section 7. Our aim in writing this code is to solve the system of equations (15)-(18) of that chapter algorithmically for a definition *d*, producing a quantity *back(d)*. This quantity gives the type of the variable defined by *d* as deduced from its subsequent uses.

As noted in Section 8 of Chapter 1, the quantity *back(d)* can best be computed by creating the tree of the program graph rooted at the block containing *d*. We shall first present a function *progtree(node)* which, when given a block, *node*, of a program graph and a global graph successor function, *cesor*, returns this tree. The tree is represented by a triple <*t*,*cont*,*succ*>, where:

t is a blank atom representing the root of the tree;

cont is a function from the blank atoms which are the nodes of the tree into the blocks of the original program graph;

succ is the multivalued descendant function of the tree.

Recall that if *p* is a program graph and *b* a block of that graph, the tree of *p* rooted at *b* is a tree whose root is *b* and such that the tree successors of a node are chosen from among its graph successors, but with avoidance of cycles. The function *progtree* sets the root of the tree to a blank atom *t* and initializes *cont(t)* to *node*. It then adds blank atoms to *succ(t)* corresponding to the elements in

cesor(cont(t)); as long as this does not create a cycle. An auxiliary function *tpred* is kept, which for each tree node returns the set of all its predecessors (including itself).

```
definef progtree(node);
     /* cesor is global */
     succ = nl;  tpred = nl;  cont = nl;  t = newat;
     cont(t) = node;
     work = {t};  /* work is the set of leaves of the tree,
                     as so far constructed */
     tpred(t) = {t};
     (while work ne nl) tnode from work;
       succ(tnode) = nl;
       (∀c∈cesor(cont(tnode)))
          /* check if adding node c to succ(tnode)  would
             cause a cycle */
          if c n ∈ cont[tpred(tnode)] then /* if not */
               /* create a new tree node */
               b = newat;  cont(b) = c;
               /* propagate the predecessor function */
               tpred(b) = tpred(tnode) with b;
               /* update the successor function */
               b in succ(tnode);
               b in work;  /* b is  now a leaf */
          end if;
       end ∀;
     end while;
     return <t,cont,succ>;
end progtree;
```

Recall that equations (15)-(18) of Chapter 1, Section 7 are to be applied to the program graph G_d divided at a definition d and not to the program graph G originally given. Accordingly, the routine *progtree* should in principle be applied to G_d and not to G. However, as already pointed out, such an approach would be prohibitively inefficient since it would mean recalculating a new program graph for each definition.

Our original motive for introducing the auxiliary object G_d in Chapter I was simply to regularize the discussion by making it possible to treat all uses appearing within a block uniformly. However, instead of actually introducing the new graph G_d, we can simply use the graph G, but in processing a definition d, calculate as if G_d had been introduced. To do this, we establish two distinct subsets among the uses of d appearing in $b(d)$. The first set, which we shall call *before(d)*, are those uses appearing prior to d in $b(d)$; the second set, which we shall call *after(d)*, are those uses appearing after d. The elements of *before(d)* (resp. *after(d)*) correspond precisely to those uses which would belong in the divided graph G_d to the first (resp. the second) of the two blocks into which the block $b(d)$ would be divided. Let *backtype(u)* be the function which determines the type of a use u from the operator applied to it and the type of the resulting object (see Chapter I, Section 7), and let orm be the auxiliary function defined by

```
definef a orm b; return if a eq Ω then b else a; end orm;
```

Then in processing the divided graph G_d the type information which we would associate with the first of the two blocks into which $b(d)$ would be divided is simply

```
prevtyp  = [con: u∈before(d)] backtype(u) orm tg.
```

Thus we can compute $back(d)$ without constructing G_d as follows (cf. Chapter I, Section 8). Given a definition d, we construct the tree t rooted at $b(d)$ from the original program graph. In using equations (15)-(18) to pass type information up t, we check each node n to see if n is a graph predecessor of $b(d)$, the root node of t. That is, we calculate the boolean quantity $blk(d) \in cesor(cont(n))$. If this has the value true, $prevtyp$ must be disjoined with the type information calculated from the tree successors of n.

We now present the recursive SETL routine $passtype$ which is responsible for passing type information up the tree t. It is called with two arguments; the first, d is the definition whose type is being determined; and the second, t is a node of the program tree at whose entry we wish to determine the type of d.

In the routine which follows, we use the macro dub defined by: `+* dub(d,b) = {u ∈ du(d)|blk(u) eq b} **` . This macro returns the set of all those uses which are chained to a definition d and which appear in a block b.

```
definef passtype(d,t);
     /* cont, succ, prevtyp, progrph, cesor, exits are
        assumed to be global */
     node = cont(t);  /* node is the basic block
               corresponding to the tree node t */
     blktyp = /* the type of d as determined from its
               uses within node  */
          [con: u∈dub(d,node)] backtype(u) orm tg;
     if node ∈ exits  /* node is a program exit */
          or /* node contains a redefinition of d's variable */
               ∃ dftup(j) ∈ progrph(node)|dftup(1) eq d(1)
          then return blktyp;
     /* in accordance with Chapter I, (15) and (17) */
     end if;
     val = dis(if blk(d) ∈ cesor(node) then prevtyp else tg,
               [dis: sb ∈ succ(t)] passtype(d,sb) orm tg);
          /* in accordance with Chapter I, (16) and the
             preceding discussion */
     return con(blktyp,val);  /* chapter I, (17) */
end passtype;
```

We now present the routine *back(d)* It creates the program tree rooted at the block of *d*, calculates *prevtyp* and calls on *passtype*.

```
definef back(d);
     /* prevtyp, cont, succ, exits, progrph, cesor are
        assumed global */
     <t,cont,succ> = progtree(blk(d) is node);
     bluses = dub(d,node); /* uses of d appearing in node */
     before = /* uses appearing before d */
              {u ∈ bluses|blkpos(u) le blkpos(d)};
     after  = /* uses appearing after d */
              bluses - before;
     prevtyp = [con: u ∈ before] backtype(u) orm tg;
     blktyp = [con:  u ∈ after]  backtype(u) orm tg;
     /* note that in calculating blktyp  we only consider
     those uses appearing in after */
     if node ∈ exits /* node is a program exit */
          /* note that here we do not have to worry about
             a redefinition since if one occurred, no uses
             in any other block could be chained to d */
          then return blktyp; /* Chapter I, (15) and (17) */
     end if;
     val = dis(if blk(d) ∈ cesor(node) then prevtyp else tg,
          [dis: sb ∈ succ(t)] passtype(d,sb) orm tg);
     /* Chapter I, (16) */
     return con(blktyp,val);  /* Chapter I, (17) */
end back;
```

CHAPTER IV

CONCRETE ALGORITHMICS OF THE TYPEFINDER

In this chapter, we shall suggest methods for coding the typefinder in languages of lower level than SETL when it is to be used in a practical compiler system. In particular, we shall describe the realization of the typefinder in two medium level languages -- BALM and PL/I. A BALM version but not a PL/I version has actually been implemented.

1. The Data Structures in the SETL Version.

In order to contrast the SETL implementation of the typefinder with lower level language implementations of this same algorithm, we first enumerate the data structures and control flow mechanisms used in the SETL implementation. Then we describe the problems which must be faced in transcribing these structures and mechanisms into BALM and PL/I. In describing these concrete algorithmic issues, we will also point out features of the medium level languages which enhance or hinder the clarity and efficiency of the resulting code.

a) The function *progrph* (Chapter III, Section 1) is used in the SETL implementation to map each basic block into a tuple of defining tuples, where each defining tuple represents a definition. From the basic mapping *prograph*, preprocessing

creates the set *defs* (Chapter III, Section 1). This set *defs* represents the set of all definitions in the program, where each definition (i.e., each element of *defs*) is represented by a triple consisting of an output variable, a block and a position within that block. The set *defs* is our initial workset and is the domain of the mapping *typ* which the typefinder aims to construct. In processing a definition *d* to determine its type by the forward and backward methods (Chapter III, Sections 4 and 5 respectively) we must often retrieve the defining tuple which *d* represents. To this end the SETL typefinder uses the macros

```
+* blk(d) = d(2) **  /* returns the block of definition  d */
+* blkpos(d) = d(3)  /* returns the position of d within blk(d)*/
+* deftup(d) = progrph(blk(d))(blkpos(d)) **
                     /* returns the defining tuple of d */
```

Note that retrieval of the defining tuple *dtup* from a definition requires four indexing operations and that retrieval of a component from *dtup* requires yet another indexing operation. Since these retrievals are all frequent operations, we must aim to implement them efficiently in lower level versions of the typefinder.

b) Another group of SETL data structures, specifically the mappings *ud* and *du*, appear in connection with use-definition chaining (Chapter III, Section 1). The first,

ud, is a mapping from the set of uses into the power set of *defs + cons* (*cons* is the set of constants) and produces the set of definitions or constants which are chained to a particular use. Note that use *u* is represented in much the same way as a definition; i.e. as a tuple whose first component is the variable used, whose second component is the block containing *u*, whose third component is the position within that block of the defining tuple which contains *u*, and whose fourth component is the position of *u* among the uses within the defining tuple of *u*. The macro *deftup* is used to retrieve the defining tuple of a particular use.

The mapping *du* is the inverse of *ud*. For each element *d* of *defs*, *du(d)* is the set of uses chained to *d*. When we apply one of the mappings *ud* or *du* we index a set by a tuple. The SETL system is designed to maximize the efficiency of such use of sets *m* as mappings. This is achieved by storing sets of tuples as hashtables and hashing the index *i* appearing in a reference *m(i)* to produce the position of the functional value.

To realize the functions *ud* and *du* in a lower level language we must aim to set up a data structure in which the functional values *ud(u)* and *du(d)* can be retrieved by a single lookup.

c) For each basic block *b* the function *cesor* returns the set of successor blocks of *b* in the program graph. Like *ud* and *du*, *cesor* is a multivalued function. It can be

realized efficiently if we represent basic blocks by successive integers and the map *cesor* itself by a tuple. The objects *entry* and *exits* which are the entry block and set of exit blocks of the program can be represented by an integer and a set of integers respectively.

d) The last major data structure which appears in the SETL implementation of the typefinder is the program tree rooted at a given node. The tree rooted at a given block *b* is returned by the coded routine *progtree(b)* (Chapter III, Section 5). Each node of the tree is represented by a blank atom on which two functions, *cont* and *succ*, are defined. The function *succ* is a multivalued tree successor function which returns the set of successor nodes of a given node. If *n* is a tree node, *cont(n)* returns the program block which *n* represents.

In the SETL implementation, *progtree(blk(d))* is called each time a definition *d* is to be processed. This is clearly inefficient; it is better to precalculate a data structure *progtree2* which collects together all the trees which would be returned by repeated individual calls to *progtree*.

e) The scheme by which type symbols are represented in the SETL implementation was discussed in Chapter III, Section 2. Elementary types are represented by bit strings so that the disjunction and conjunction of elementary types are calculation by 'and'ing and 'or'ing their representative bit

strings. Complex types are represented by tuples whose first component is an integer representing the grosstype, whose second component is a bit string representing a possible elementary alternand, and whose remaining components represent the constituent components of the complex type symbol.

For each definition *d* of the program, the function *typ* defined on *defs + cons* returns the type symbol representation momentarily assigned to the output of *d*; for a constant *c*, *typ(c)* gives the type symbol representation for *c*.

2. Implementing the Typefinder in BALM.

Two problems must be overcome in translating SETL programs into a lower level language. The first concerns the representation of sets since in the SETL typefinder we often deal with sets. For example, *ud(u)* is a set of definitions while *du(d)* is a set of uses. The programmer can use a variety of data structures to represent sets in lower level languages. Among these are arrays, lists and bit strings. Arrays are most useful when the sets they represent have a fixed number of elements, since in such a case the entire array can be allocated at once and the values

filled in as objects are created and put in the set. In the more common case of a set whose cardinality is unknown but which grows to some final size, a list representation is more useful since objects can be added to the head of the list as they are added to the set. Note, however, that arrays and lists are only convenient for sets which are built up and then used as index sets to control some iteration. In such cases, one iterates over the array or list performing some process once for each member. However, arrays and lists are unsuitable for membership testing since such testing will often require a search through most of the list or array. Of course one can implement a set as an array using hashing but this type of code, used internally within SETL, is not easy for a programmer to manage.

An alternative representation of a set S is a bit string. Such a representation can be used when S is known to be a subset of some stable larger set S' whose elements can be put into a one-to-one correspondence with the integers from 1 to $\#S'$. The set S can then be represented by a bit string B, where $\#B = \#S'$. An element $a_i \in S'$ is a member of S if and only if $B(i) = 1$. The bit string representation has the advantage of wasting only one bit for each absent element; often it will more than make up this small space loss by not replicating the individual elements of the superset S'. Bit string representations are most useful in dealing with highly volatile sets for which insertions and

deletions are constantly being performed. To insert an element $a_i \in S'$ into S one merely sets $B(i)$ to 1; to delete a_i, one merely sets $B(i)$ to 0. This may be contrasted with the way in which the same operations must be carried out when a list representation is used. To insert an element a in a set represented as a list, one must check each element of the list to see if a is not already a member. Membership testing is also very efficient for sets represented as bitstrings. However, the bit string representing can lead to an inefficient realization of iterations over S, especially if S is sparse with respect to S'. In such cases, many 0 bits must be passed over before one finds a member a_i of S' which is also a member of S.

A second problem which must be faced in translating SETL programs into lower level languages is that of realizing the operation of indexing by compound objects. For example, in the SETL typefinder, uses and definitions are represented by tuples. Such tuples appear as indices to the mappings ud and du. However, in BALM the only indexable object is the vector, and vectors can be indexed only by integers. As we shall see, this forces us to represent uses and definitions in BALM by integers. These integers are used to index two vectors, UD and DU, and also to index a vector DEFS which defines the actual definition which corresponds to each 'definition representing' integer. This 'cross-indexing' of integers to put them in correspondence with compound data

objects is a fundamental device of lower level programming.

Having made these general remarks, we turn to a specific discussion of the data structures used in the BALM implementation and of their relation to the corresponding SETL data structures.

a) A definition in the SETL implementation is represented by a triple consisting of an output variable, a block number and a position-within-block number. The latter two data are used as indices to the structure *progrph* which contains all defining tuples. Concerning the SETL typefinder, we have noted that given a definition *d*, retrieval of the operator *op(d)* required five indexing operations. This is clearly unsatisfactory in an efficient implementation.

To increase efficiency in the BALM implementation, we eliminate the object *progrph* entirely and incorporate all information about the definitions of the program into a vector DEFS. Each component of DEFS is itself a vector representing a defining tuple, and consists of the following items:

1) an integer representing the basic block in which the definition appears
2) the variable being defined
3) an integer representing the operator
4) the input variables.

Note that it is no longer necessary to store the position of a definition within its basic block, since

definitions are ordered within the vector DEFS according to their appearance in the program.

Given the data object DEFS, a definition is represented by its integer index within DEFS. Given an integer I, the defining tuple of definition I is recovered by accessing DEFS[I] and the operator is recovered by DEFS[I] [3]. Note that the number of indexing operations necessary to recover an operator from a definition has been reduced from five to two.

The advantage of using the SETL object *progrph* was that, given a basic block *b*, the expression *progrph(b)* accessed the set of definitions appearing within *b*. If only the vector DEFS were available, DEFS would have to be searched sequentially to find an I such that DEFS[I][1]=B. To avoid this and to enable quick access to the set of definitions appearing within a basic block, we introduce two cross indexing functions -- LODEF and HIDEF. If B is an integer representing a basic block, LODEF[B] is the index (to DEFS) which represents the first definition within block B and HIDEF[B] is the index which represents the last definition within B.

In a similar way, uses are represented by integers. These integers can be considered as indices to a virtual data object USES (because of the availability of other cross indices, an actual data object USES is unnecessary). For each use U, UDEF[U] gives the definition to which U serves as input.

If D is an integer representing a definition, LOUSE[D] is the index (to USES) which represents the first argument to the operation defining D and HIUSE[D] is the index which represents the last argument to this operation.

b) We noted earlier that it is desirable to implement the functions *ud* and *du* in a way that allows their functional values to be retrieved by a single lookup. Given that we are using an integer encoding of uses and definitions, this goal can be achieved by introducing two vectors UD and DU. The uses which are chained to a definition D can then be accessed by DU[D] and the definitions chained to a use U can be accessed by UD[U]. However, the question of how to represent the values DU[D] and UD[U] reamins. Since these sets do not change during the execution of the typefinder routine, and since they are primarily used for iteration, we choose to represent them as BALM lists. This representation is not the best for use during the use-definition chaining process when the objects UD and DU are being created and are constantly involved in membership tests. However, since BALM does not provide bit strings of arbitrary length and since the overhead of simulating long bit strings (using vectors of integers) turns out to be substantial, we use lists to represent UD[D] and DU[D] even during the chaining phase.

c) The block successor function *cesor* is also represented by a vector of lists. Thus if B represents a block, CESOR[B] is the list of integers which represent the successor blocks

of B. ENTRY is the entry node to the program and EXITS a list of exit nodes.

d) Instead of the coded routine *progtree* which is called each time the program tree rooted at a given node is needed, the BALM implementation provides a vector PROGTREE which is indexed by integers representing basic blocks. This vector is created once at the start of the typefinding process.

A program tree T is represented by a list whose first element is an integer representing the block B which is the root of T and whose other elements are lists representing the subtrees rooted at the successors of B. Thus, the SETL *cont(t)* corresponds to the BALM HD T and the SETL *succ(t)* corresponds to TL T.

To illustrate the consequence of the concrete algorithmic design which has just been sketched, we present a BALM version of the SETL routine *passtype* (Chapter III, Section 5). This routine is responsible for passing type information up a program tree. The routine is annotated using PL/I - SETL style comments (/* THIS IS A COMMENT */) although the actual BALM comment convention is different.

```
PASSTYPE = PROC(D,T), /* D IS A DEFINITION, T A PROGRAM TREE */
     BEGIN(NODE,BLKTYP,DUD,UDFL,OUTVBL,K,SUCC,L),
     NODE = HD T /* ROOTNODE OF T */
```

```
BLKTYP = TG /* CORRESPONDS TO THE orm tg IN THE
          SETL VERSTION */,
DUD = DU[D] /* SET OF USES CHAINED TO D */
WHILE DUD REPEAT DO /* ITERATE OVER SET OF USES */
     L = HD DUD /* A PARTICULAR USE */,
     DUD = TL DUD /* UPDATE SET FOR FURTHER ITERATION */,
     DF = DEFS[UDEF[L]] /* THE DEFINING TUPLE WHICH
                        CONTAINS THAT USE */,
     IF DF[1] EQ NODE THEN /* THE USE APPEARS WITHIN
                        THE TREE ROOT NODE */
          BLKTYP = CON(BLKTYP, BACKTYPE(L))
     /* IN ACCORDANCE WITH CHAPTER I, (17) */
END,
IF MEMBER(NODE, EXITS) /* THIS MEMBERSHIP TEST WILL BE
                        INEFFICIENT */
     THEN /* NODE IS A PROGRAM EXIT */
          RETURN BLKTYP,
OUTVBL = DEFS[D][2] /* THE DEFINED VARIABLE OF D */,
FOR K = (LODEF[NODE], HIDEF[NODE]) REPEAT
     /* ITERATE OVER ALL DEFINITIONS IN THE BLOCK */
     IF DEFS[K][2] EQ OUTVBL THEN
          /* THE BLOCK CONTAINS A REDEFINITION OF
                        D's VARIABLE */
          RETURN BLKTYP,
SUCC = TL T /* SET OF TREE SUCCESSORS */
```

```
     VAL = IF MEMBER (DEFS[D][1], CESOR[NODE]) THEN
          PREVTYP ELSE (IF SUCC THEN TZ ELSE TG),
     /* SEE CHAPTER I, (16) AND THE DISCUSSION IN
          CHAPTER III, SECTION 5 */
     WHILE SUCC REPEAT DO
          /* ITERATE OVER TREE SUCCESSORS */
          IF VAL EQ TG THEN RETURN BLKTYP
          /* NO TYPE INFORMATION MAY BE OBTAINED FROM
          THE SUCCESSOR NODES SINCE THERE IS A PATTERN
          OF USES ALONG SOME PATH WHICH IMPLIES TYPE TG */,
          L = HD SUCC,  SUCC = TL SUCC,
          VAL = DIS(VAL, PASSTYPE(D,L))
     END,
     RETURN CON(BLKTYP,VAL) /* CHAPTER I, (17) */
END END;
```

Note that in the above BALM routine we have explicitly incorporated an optimization which was not used in the SETL version. As soon as it is detected that the uses along some subtree imply a result type *tg*, we return from PASSTYPE without bothering to see what happens along the other subtrees.

e) Next we address the question of type representation in BALM. Using the BALM vector capabilities, types can be represented in much the same way that they are represented in SETL. Elementary types are represented by BALM bit strings

(actually, integers interpreted as bit strings) and grosstypes are represented by such BALM objects as [2 0 TI] for $\{ti\}$, [3 TT TB] for $[tb]|tt$ and [4 TI TI TC [2 0 TI]] for $<ti,tc,\{ti\}>|ti$. This allows us to use BALM's vector creation and concatenation operations to construct type representations.

To illustrate the concrete algorithmic situation which results, we present the BALM routine which forms the conjunction of two type symbols. This routine is modeled after the SETL routine *con* of Chapter III, Section 2. Note the use of the BALM built-in function LAND which, given two integers I and T, returns an integer whose binary representation is the boolean conjunction of the binary representations of I and J.

```
CON = PROC(A,B),
     BEGIN(ELA,ELTYP,I,TUP,CONJ),
     IF GROSTYP(A) GT GROSTYP(B) THEN RETURN CON(B,A),
     IF ISEL(A) AND ISEL(B) THEN /* 2 ELEMENTARY TYPES */
          RETURN LAND(A,B),
     ELA = /* ELEMENTARY ALTERNAND OF A */
          IF ISEL(A) THEN A ELSE A[2];
     ELTYP = /* THE ELEMENTARY ALTERNAND OF THE RESULT */
          LAND(ELA, B[2]),
     IF ISKNT(B) THEN DO TUP = VECTOR(KNT,ELTYP),
          IF ISUNT(A) THEN /* CONJUNCTION OF UNKNOWN
                              AND KNOWN LENGTH TUPLES */
```

```
            DO TUP = CONCAT(TUP,MAKVECTOR(SIZE B-2)),
                FOR I=(3,SIZE B) REPEAT DO
                    CONJ = CON(B[I],A[3]),
                    IF CONJ EQ TZ THEN RETURN ELTYP,
                    TUP[I] =  CONJ
                END
            END,
        IF ISKNT(A) AND SIZE A EQ SIZE B THEN
            /* 2 TUPLES OF SAME LENGTH */  DO
            TUP = CONCAT(TUP,MAKVECTOR(SIZE B-2)),
            FOR I = (3, SIZE B) REPEAT DO
                CONJ = CON[A[I], B[I]),
                IF CONJ EQ TZ THEN RETURN ELTYP,
                TUP[I] = CONJ
            END
        END,
        RETURN TUP
    END,
    IF GROSTYP(A) EQ GROSTYP(B) THEN
        /* 2 SETS OR TUPLES OF UNKNOWN LENGTH */
        DO CONJ = CON(A[3], B[3]),
            IF CONJ EQ TZ THEN RETURN ELTYP,
            RETURN VECTOR(A[1], ELTYP, CONJ)
        END,
    RETURN ELTYP
END END;
```

3. The Typefinder in PL/I.

For programming the typefinder, BALM has several advantages over PL/I. BALM allows its user to concatenate two vectors explicitly while in PL/I he must allocate the necessary storage and manually copy the information.

BALM also allows its user to allocate vectors of dynamically determined size without specific declaratory information becoming necessary. It also makes it convenient to use "jagged" arrays; that is linear arrays each of whose elements is itself an array of undetermined length. In PL/I of course, the size of the second dimension of a two-dimensional array is the same along each cross-section of the first dimension; though PL/I does allow the "jagged" effect to be obtained using pointers. In BALM, vectors and lists are maintained in a garbage collected environment, which frees the programmer from the problem of keeping track of all his pointers and explicitly releasing storage when an object is no longer necessary.

In the present section we shall explain how the data structures needed in the typefinder can be represented in a PL/I version, and also write some samples of PL/I typefinder code.

a) As in BALM, definitions and uses will be represented by integer indices into an object DEFS and a virtual object USES. In BALM, DEFS was a vector of defining vectors. Each defining vector contained a block number, an output variable,

an operator and several input variables. In PL/I, we simulate "jagged" arrays using structures and pointers. The object DEFS can be declared as follows:

```
DCL  1  DEFS(NUMDEFS),
        2 BLOCK FIXED,
        2 OUTVBL PTR,
        2 OP FIXED,
        2 INVBLS PTR;
```

OUTVBL is a pointer to the symbol table entry which represents the defined variable of a definition; INVBLS is a pointer to a list each of whose elements is declared by

```
DCL  1  LISTVBL BASED(P),
        2 VBL PTR,
        2 NEXT PTR;
```

VBL is a pointer to the symbol table and NEXT is a list pointer to the next LISTVBL in the same list.

As in BALM, the arrays LODEF and HIDEF will be used to access the first and last definitions of a block; LOUSE and HIUSE will be used to access the first and last uses input to a definition and UDEF will specify the definition to which each particular use is an argument.

b) The functions *ud* and *du* can be declared by

```
DCL(UD(NUMUSES), DU(NUMDEFS)) PTR;
```

That is, UD and DU are arrays of pointers. DU(D) and UD(U) point to objects of the form declared by

```
DCL  1  LISTINT BASED(Q),
        2 INTEG FIXED,
        2 NEXT PTR;  .
```

Such a representation is well suited for the typefinder, since DU(D) and UD(U) are used primarily as bases for iteration. Note that, unlike the current version of BALM, PL/I provides 'long' bit strings. During the use-definition chaining process, in which the objects DU and UD are constructed it is more efficient to represent DU and UD by arrays of bitstrings. These would be declared by

```
DCL  UDBIT(NUMUSES) BIT(NUMDEFS + NUMCONS),
     DUBIT(NUMDEFS) BIT(NUMUSES);
```

In passing from use-definition chaining to typefinding, the arrays UD and DU can be created from UDBIT and DUBIT.

c) In the typefinder, the function *cesor* is used primarily for membership testing but must support iteration when we build the program trees from the program graph. A reverse pattern of usage is seen in the use-definition chaining algorithm, in which *cesor* is used primarily for iteration although some membership testing is needed. Thus for efficiency reasons, it probably pays to keep *cesor* both as an array of lists and bitstrings. We declare CESOR by

```
DCL  1  CESOR(NUMBLKS),
        2 BITCES BIT(NUMBLKS),
        2 LISTCES PTR;
```

Then LISTCES will point to a structure of the form previously declared as LISTINT. ENTRY will be an integer and EXITS, used exclusively for membership testing, will be a bitstring. These objects are declared by

```
DCL ENTRY FIXED, EXITS BIT(NUMBLKS);  .
```

d) In PL/I, PROGTREE will be an array of pointers declared by

```
DCL PROGTREE(NUMBLKS) PTR;  .
```

Each element of PROGTREE will point to a structure declared as follows:

```
DCL  1  TREE BASED(R),
        2 NODE FIXED,
        2 YOUNGER_BROTHER PTR,
        2 OLDEST_SON PTR;  .
```

Both YOUNGER_BROTHER and OLDEST_SON will point to structures declared as TREEs.

Having reviewed these declarations, we can now give an example of a PL/I routine which might be used in the type-finder. The following function PASSTYPE is a PL/I version of the SETL *passtype* (Chapter III, Section 5) and the BALM PASSTYPE (Chpater IV, Section 2). We assume it to be internal to a block in which all of the preceding declarations have been made. PASSTYPE returns a pointer to a type structure (see the following subsection), as do the functions CON, DIS and BACKTYPE. TG , TZ and PREVTYP are global pointers to type structures representing *tg*, *tz* and *prevtyp*.

```
PASSTYPE: PROC(D,T) RECURSIVE RETURNS(PTR);
     DCL D FIXED, /* D is an integer representing a definition*/
          T PTR, /* T points to a TREE  */
          (BLKTYP,VAL) PTR, /* pointers to type structures */
          OUT PTR, /* pointer to symbol table */
          DUD PTR, /* pointer to list of LISTINTs */
          SUCC PTR, /* pointer to a TREE */
          (L,K,DF) FIXED; /* represent definitions or uses */
     BLKTYP = TG; /* corresponds to orm tg in the SETL version */
     DUD = DU(D); /* ptr to list of uses chained to D */
LOOP: /* iterate over set of uses chained to D */
     DO WHILE(DUD ¬= NULL);
          L = DUD → INTEG;  /* get the next use from the list */
          DUD = DUD →  LISTINT.NEXT;  /* update pointer for
                         further iteration */
          DF = UDEF(L);  /* the definition in which the use
                                          L appears */
          IF BLOCK(DF) = T → NODE THEN
               /* the use appears in the root node of the tree */
               BLKTYP = CON(BLKTYP, BACKTYPE(L)); /*Ch.I, (17)*/
     END LOOP;
     IF SUBSTR(EXITS, T→NODE, 1) THEN
          /* NODE is a program exit */ RETURN BLKTYP;
          /* in accordance with Ch. I, (15) */
     OUT = OUTVBL(D); /* a pointer to the symbol table entry
                         of the variable defined by D */
```

```
REDEF: /* iterate over all definitions in the block,
          searching for a redefinition */
     DO K = LODEF(T → NODE) TO HIDEF(T → NODE);
          IF OUT = OUTVBL(K) THEN
               /* a redefinition has been found */ RETURN BLKTYP;
          /* in accordance with Ch. I, (15) */
     END REDEF;
     SUCC = T → OLDEST_SON;
     IF SUBSTR(BITCES(T → NODE), BLOCK(D), 1) THEN
          /* the block containing D is a successor of
               the current node */  VAL = PREVTYP;
          /* see Ch. I, (16) and the discussion in Ch. III,
                                        Section 5 */
          ELSE
               IF SUCC = NULL THEN /* no tree descendents */
                    VAL = TG; /* corresponds to orm tg
                                        in SETL version */
                    ELSE VAL = TZ; /* identity value
                                        for disjunction */
SUBTREES:  /* iterate over all subtrees */
     DO WHILE(SUCC ¬= NULL);
          IF VAL = TG THEN /* there is a pattern of uses
               along some path which implies type tg so no
               type information may be obtained from uses
               along alternate paths */
               RETURN BLKTYP;
```

```
            VAL = DIS(VAL, PASSTYPE(D,SUCC)); /* Ch. I, (16) */
            SUCC = SUCC → YOUNGER_BROTHER;
            /* update for continued iteration */
        END SUBTREES;
        RETURN(CON(BLKTYP,VAL)) /* Ch. I, (17) */
END PASSTYPE;
```

e) In BALM, the typestatus vector was represented by the BALM vector TYPE indexed by a definition D. Each TYPE[D] was either an integer representing an elementary type or a vector representing a grosstype. In the case of an object of type 'set' or 'unknown length tuple', this vector was of length 3 but in the case of an object of type 'known length tuple' the vector was of arbitrary size.

PL/I, however, does not permit arrays which are not homogeneous; that is, all the elements of a PL/I array must be of the same size. One solution is to make TYPE an array of pointers. If this is done TYPE(D) will point to an integer in the case of an elementary type and a structure in the case of a grosstype. This adds an extra level of indirection and also creates the problem of how to tell whether the target of a pointer is an integer or a structure.

An alternative is to declare the array TYPE as follows:

```
DCL  1  TYPE(NUMDEFS + NUMCONS),
        2 GROSTYP FIXED,
        2 ELTYP BIT(8),
        2 COMPONENTS PTR;  .
```

In the case of an elementary type, GROSTYP will be 1, ELTYP will contain the bitstring representation of the type and COMPONENTS will be NULL. In the case of a grosstype, GROSTYP will equal 2, 3 or 4 (depending on whether the object is of type set, unknown length tuple or known length tuple), ELTYP will be the elementary alternand and COMPONENTS will point to a list of type structures declared by:

```
DCL  1  TYPELIST BASED(S),
        2 LISTGROS FIXED,
        2 LISTEL BIT(8),
        2 LISTCOMP PTR,
        2 LISTNEXT PTR;  .
```

If TYPE(D).GROSTYP is 2 or 3, then COMPONENTS(D) will point to only a single structure of type TYPELIST; otherwise it will point to an entire list of type structures chained by the pointer LISTNEXT.

Note that the programmer will himself have to allocate and free structures of type TYPELIST. As soon as a TYPELIST is no longer needed it must be freed or the proliferation of TYPELISTs will soon devour all of available memory. This problem was entirely ignored in presenting the routine PASSTYPE above. For example, after executing the statement

VAL = DIS(VAL, PASSTYPE(D, SUCC)), the TYPELIST which was pointed to by PASSTYPE(D,SUCC) must be freed. A similar situation exists in the statement BLKTYP = CON(BLKTYP, BACKTYPE(L)). The TYPELISTs to which the old values of VAL and BLKTYP were pointing must also be freed.

The fact that it is necessary to free TYPELISTs causes a problem with the objects TG and TZ. We assumed in programming PASSTYPE that TG and TZ were constant pointers to type structures representing *tg* and *tz* respectively. However, if TG or TZ is returned by a routine which results in a type value and the TYPELISTs to which TG and TZ point are freed, then we can no longer use TG and TZ to represent *tg* and *tz*. To solve this problem, we will declare TG and TZ as structures as in the following way:

```
DCL  1  TG,
        2 TGGROS FIXED INIT(1),
        2 TGEL BIT(8) INIT('11111111'B),
        2 TGCOMP  PTR INIT(NULL),
        2 TGNEXT PTR INIT(NULL);  and
DCL  1  TZ,
        2 TZGROS FIXED INIT(1),
        2 TZEL BIT(8) INIT('00000000'B),
        2 TZCOMP PTR INIT(NULL),
        2 TZNEXT PTR INIT(NULL);  .
```

We now present a useful auxiliary routine FR which, when given a pointer to a TYPELIST, frees all the storage occupied by that type structure.

```
FR: PROC(P) RECURSIVE;
     DCL P PTR,
     IF P → LISTNEXT ¬= NULL THEN CALL FR(P → LISTNEXT);
     IF P → LISTCOMP ¬= NULL THEN CALL FR(P → LISTCOMP);
     FREE P → TYPELIST;
   END FR;
```

Next we present a PL/I version of the routine CON which forms the conjunction of two types.

```
CON: PROC(A,B) RECURSIVE RETURNS(PTR);
     DCL(A,B) PTR, /* pointers to type structures */
        (AAUX, BAUX, RAUX,RES,CONJ,TRES) PTR,
               /* pointers to auxiliary type structures */
     IF A → LISTGROS > B → LISTGROS THEN RETURN(CON(B,A));
     ALLOCATE TYPELIST SET(RES);
          /* RES will point to a resulting elementary
                                          type structure */
     RES → LISTGROS = EL;
     RES → LISTEL = A → LISTEL & B → LISTEL;
                          /* elementary alternand of the result*/
     RES → LISTCOMP, RES → LISTNEXT = NULL;
     IF ISKNT(B) THEN  /* B is a known-length tuple, so result
                          must be a known-length tuple or
                          elementary */
```

```
BKNT: DO;
     ALLOCATE TYPELIST SET(TRES);
     /* TRES points to a structure representing
                         the resultant tuple */
     TRES → LISTGROS = KNT;
     TRES → LISTEL = RES → LISTEL;
     IF ISUNT(A) THEN /* conjunction of known-length
                         and unknown-length tuples */
     AUNT: DO;
          AAUX = A → LISTCOMP; /* component of A */
          BAUX = B → LISTCOMP; /* first component of B */
          CONJ = CON(AAUX, BAUX);
          IF CONJ → TYPELIST = TZ THEN DO;
               /* free extra storage and return RES */
               FREE CONJ → TYPELIST;
               FREE TRES → TYPELIST;
               RETURN(RES);  END;
          /* otherwise, insert first component of result */
          TRES → LISTCOMP = CONJ;
          RAUX = CONJ; /* pointer to last filled-in
                          component of result */
          OVER: /* repeat for each component of B */
               DO WHILE(BAUX → LISTNEXT ¬= NULL);
                    /* get the next component */
                    BAUX = BAUX → LISTNEXT;
```

```
          CONJ = CON(AAUX, BAUX);
          IF CONJ → TYPELIST = TZ  THEN DO;
               /* free extra storage and return*/
               FREE CONJ → TYPELIST;
               CALL FR(TRES);
               RETURN(RES); END;
          /* insert component into result */
          RAUX → LISTNEXT = CONJ;
          RAUX = CONJ; /* last filled component*/
     END OVER;
END AUNT;
IF ISKNT(A) THEN /* conjunction of two known-
                    length tuples */

AKNT:  DO;
     /* get first components of A and B */
     AAUX = A → LISTCOMP;  BAUX = B → LISTCOMP;
     CONJ = CON(AAUX, BAUX);
     IF CONJ → TYPELIST = TZ THEN DO;
          /* free extra storage and return RES*/
          FREE CONJ → TYPELIST; FREE TRES→ TYPELIST;
          RETURN(RES);   END;
     /* otherwise, insert first component
                                        of result */
     TRES → LISTCOMP = CONJ;
     RAUX = CONJ; /* pointer to last filled-in
                    component of result */
```

```
      AGAIN: DO WHILE /* both A and B still have
               components which were not processed*/
           (BAUX→ LISTNEXT ¬=NULL &
                      AAUX → LISTNEXT ¬= NULL);
           BAUX = BAUX → LISTNEXT;
                /* next component of B */
           AAUX = AAUX → LISTNEXT;
                /* next component of A */
           CONJ = CON(AAUX,BAUX);
           IF CONJ → TYPELIST = TZ  THEN DO;
                /* free extra storage and return*/
                FREE CONJ → TYPELIST;
                CALL FR(TRES);
                RETURN(RES);  END;
           RAUX → LISTNEXT = CONJ;
             /* insert component into result */
           RAUX = CONJ; /* last filled component*/
      END AGAIN;
      /* Ensure that A and B were of same length */
      IF BAUX → LISTNEXT ¬= NULL |
                         AAUX → LISTNEXT = NULL
           THEN /* unequal length so return RES*/
           DO;  CALL FR(TRES);
                RETURN(RES); END;
END AKNT;
      /* otherwise, free RES and return TRES */
```

```
                FREE RES → TYPELIST;  RETURN(TRES);
            END BKNT;
            IF A → LISTGROS = B → LISTGROS & A → LISTGROS ¬=1 THEN
                /* two sets or tuples of unknown length */
              SETSORUNTS: DO;
                CONJ = CON(A → LISTCOMP, B → LISTCOMP);
                IF CONJ → TYPELIST = TZ THEN DO;
                    FREE CONJ → TYPELIST;
                    RETURN(RES);  END;
                RES → LISTCOMP = CONJ;
                RES → LISTGROS = A → LISTGROS;
            END SETSORUNTS;
        RETURN(RES);
END CON;
```

We are now in a position to present a revised version of the routine PASSTYPE which frees storage when appropriate and which uses the revised declarations of TZ and TG. This should be compared with the previous PL/I version of PASSTYPE; the complications caused by forcing the programmer to do his own storage management stand out with reasonable clarity.

```
PASSTYPE: PROC(D,T) RECURSIVE RETURNS(PTR);
     DCL(D,L,K,DF) FIXED,
        (T, BLKTYP, VAL,OUT,DUD,SUCC,TEMP1,TEMP2) PTR;
     ALLOCATE TYPELIST SET(BLKTYP);
     BLKTYP → TYPELIST = TG;
     DUD = DU(D);
     LOOP:  DO WHILE(DUD ¬= NULL)
          L = DUD → INTEG;  DUD = DUD → LISTINT.NEXT;
          DF = UDEF(L);
          IF BLOCK(DF) = T → NODE THEN DO;
               TEMP1 = BLKTYP;
               TEMP2 = BACKTYPE(L);
               BLKTYP = CON(TEMP1,TEMP2);
               CALL FR(TEMP1);
               CALL FR(TEMP2);   END;
     END LOOP;
     IF SUBSTR(EXITS, T → NODE, 1) THEN RETURN(BLKTYP);
     OUT = OUTVBL(D);
   REDEF: DO K=LODEF(T → NODE) TO HIDEF(T → NODE);
          IF OUT = OUTVBL(K) THEN RETURN(BLKTYP);
     END REDEF;
     SUCC = T → OLDEST_SON;
```

```
    IF SUBSTR(BITCES(T → NODE), BLOCK(D), 1) THEN VAL=PREVTYP;
         ELSE DO;
              ALLOCATE TYPELIST SET(VAL);
              IF SUCC = NULL THEN VAL → TYPELIST = TG;
                             ELSE VAL → TYPELIST = TZ;
              END;
  SUBTREES: DO WHILE (SUCC ¬= NULL);
    IF VAL → TYPELIST = TG THEN RETURN(BLKTYP);
    TEMP1 = VAL;
    TEMP2 = PASSTYPE(D,SUCC);
    VAL = DIS(TEMP1,TEMP2);
    CALL FR(TEMP1);
    CALL FR(TEMP2);
    SUCC = SUCC → YOUNGER_BROTHER;
  END SUBTREES;
  TEMP1 = VAL;
  VAL = CON(TEMP1,BLKTYP);
  CALL FR(TEMP1);
  CALL FR(BLKTYP);
  RETURN(VAL);
END PASSTYPE;
```

4. Summary and Conclusions.

Several conclusions may be drawn from a comparison of the SETL version of the typefinder and the two more concrete versions which have been presented in the preceding pages. We note in the first place that the stepwise translation of complex algorithms into successively lower level languages is a valuable programming tool. The SETL version of the typefinder shows the basic features of the algorithm with reasonable clarity and is not badly cluttered with irrelevant detail. For this reason, SETL was found to be a convenient language for initial development of the algorithm.

Once a SETL algorithm is debugged and running, its transcription into BALM is a straightforward process. The key problem to be faced in this transcription is how to represent all necessary SETL data objects in BALM. This decision must be made by noting how each SETL data structure is used in a particular algorithm and by deciding how such uses can be mimed most efficiently in BALM. Once these decisions are made, the actual coding of the BALM routine is a simple matter. During such coding, numerous optimizations possible at the lower level of language will turn up. The transcription of BALM code into PL/I may be described in similar terms.

We may also note the convenience of garbage collection in developing an algorithm such as the typefinder. One argument against garbage collection is that it is unnecessarily expensive and that the programmer can free data structures himself when the phase of an algorithm which uses these data structures is completed. This is true if a few large data structures are used continually in some identifiable subsection of an algorithm to produce specific results which are used in the next subsection. However, in an algorithm like the typefinder, compound type symbols are being created and discarded constantly. These type symbols are not part of a single large data structure which can be allocated and freed as a whole. Rather, each type symbol is a small independent data structure which must be allocated and freed independently. The complications forced on the programmer if he must manage storage manually are large and detract from his ability to solve the problem neatly and efficiently.

We note that it is doubtful whether a PL/I algorithm for type determination could have been developed from the SETL algorithm with even a small fraction of the ease with which it was developed from the BALM version. This is because in going from SETL to BALM to PL/I, the process of transcription is broken up into two parts; first one transcribes data structures, then one introduces storage management. This process of stepwise refinement is invaluable

in transforming a complex (SETL) algorithm into a version of itself written in a language (PL/I) of very substantially lower level.

CHAPTER V

MODIFICATIONS AND EXTENSIONS OF THE TYPEFINDING ALGORITHM

1. Removing Type I Imprecisions

In Chapter II, Section 7 we defined a type I imprecision as occurring when an object is incorrectly identified as possibly being of type *tu* -- the undefined atom. Such an imprecision typically arises when a function *f* is being indexed by an element *x* which must belong to the domain of *f* but the typefinder is unable to detect this membership. An example of this, noted in Chapter II, is the statement

```
(1 ≤ ∀k ≤ #i) intov(i(k)) = i;;
```

for which the typefinder classifies *i(k)* as being of type *ti|tu* despite the fact that *i* is known to be of type *[ti]|tt* and *k* is clearly in range.

The second method of type determination can often establish that an object is not of type *tu* by examining the subsequent uses of the object. For example, in the statement

```
if wfreq(x) lt least then <keep,least> = <w,wfreq(x)>;;
```

the first occurrence of *wfreq(x)* is recognized as being of type *ti* since it is used as an operand to a comparison operator while the undefined atom may not be used in such a manner. However, the second occurrence of *wfreq(x)*, which

is not used in this way, can only be identified as being of type *ti|tu*. Of course, in a compiler incorporating an algorithm to eliminate redundant computations, this imprecision would be removed.

Let us reconsider our first example. Even without using the fact that *k* is within the domain of *i* we can show that *i(k)* cannot be Ω since *i(k)* is used as an index to the mapping *intov* and Ω cannot be used in this manner. This fact should be detectable by our second method. It is not detected only because of the coarseness of our type calculus. Since almost any object, including a set, tuple, integer, bit string, etc., can serve as an index to a mapping, very little information can be gleaned from the fact that a variable is used as an index. The disjunction of all the disparate types which can be indices is *tg*; however an object used as an index cannot be of type *tu*.

To avoid losing the fact that an index cannot be Ω, we should introduce a new type symbol *td* to designate "some SETL object not equal to Ω." The disjunction of a set and a tuple would then no longer be *tg* but *td*. In the preceding example, if *u* represents the use of *i(k)*, then *backtype(u)* would be *td* rather than *tg*; the presence of such a symbol in our type calculus would then allow us to identify *i(k)* as *ti* rather than *ti|tu*.

Introduction of the symbol *td* would somewhat complicate the formation of conjunctions and disjunctions. It remains

to be seen on an experimental basis whether the added information which could be obtained is worth these complications.

Some type I imprecisions arise in connection with loops containing type II imprecisions. The following loop exemplifies this:

```
(while seen ne nℓ) node from seen;
                    ⋮
end while;
```

Although *seen* cannot equal nℓ within the loop, this fact is not detected by our algorithm, so that *seen* is incorrectly identified as of type $\{ti\}|tn$. This is a type II imprecision. Since *node* is an arbitrary element of a possibly null set, its type is taken as $ti|tu$ leading to a type I imprecision. Moreover, at the point immediately after the *while* loop, *seen* must be of type *tn* but this fact is also missed by the typefinder.

As the following example indicates, a similar situation can arise in "forall" loops:

```
(∀z ∈ newin) int(#int+1) = z; ...
```

The loop header is translated into a test of the condition *newin* eq nℓ; if the test fails *z* is extracted and the loop body executed. However, the typefinder does not detect the fact that when the loop body is executed, *newin* cannot be nℓ. For this reason, *z* is imprecisely classified as being of type $ti|tu$.

A solution to this problem is to have the compiler automatically insert dummy statements after all loop headers to make explicit the logical type conditions which those headers imply. This can be done by assigning constants of known type to variables or by defining variables using operators which will produce the correct types. These dummy statements would of course not be executed but would exist only in the compilation phase to aid in the type determination process.

2. Mappings of One Argument

In Chapter II it was pointed out that subroutine parameters are treated by the typefinder as items of general type. The only way to identify the types of such objects, barring a general cross-subroutine type determination process, is by their subsequent use; i.e., by the second method of type determination.

Parameters p which are themselves mappings will generally be used only in function applications of one of the forms $p(x)$, $p\{x\}$, etc. If only the form $p(x)$ occurs and x may be an integer, the question of whether p represents a tuple or a set is left indeterminate. Indeed, this indeterminacy may be intended by the programmer, who may wish to keep open the final choice of a data structure to represent some one of the mappings which he uses. Unfortunately, however, the disjunction of the set type and the tuple type in our calculus

is *tg* -- the general type. In our typefinder, parameters representing mappings are therefore apt to be classified as being of type *tg*, so that even the fact that the parameter is a mapping will be lost.

In treating the examples described in Chapter II, we avoided this difficulty by explicitly declaring the types of all parameters. However, it is desirable to detect the fact that an object is a mapping without having to resort to declarations.

As a first step toward this goal, we can introduce a new type symbol representing a 'mapping of one argument'. The following definition makes explicit one way in which this can be done.

Definition: Let *t1* be a type symbol such that *con(ti,ti) = ti* and let *t2* be any type symbol. Let *nl(t1)* < 3 and *nl(t2)* < 3. Then the symbol *(t1,t2)* designates "a one-argument mapping from items of type *t1* to items of type *t2*".

The type *(t1,t2)* is meant to serve as the disjunction of the type symbols {<*t1,t2*>} and [*t2*]. (Note that [*t2*] itself is the disjunction of the symbols <*t2*>,<*t2,t2*>,<*t2,t2,t2*>,...). Note the restriction *con(ti,ti) = ti* that appears in the preceding definition. When this condition fails to hold then a mapping from objects of type *t1* to objects of type *t2* will be recognized as being of type {<*t1,t2*>} since a tuple can only be indexed by an integer.

To add to our calculus the new grosstype which appears in the preceding definition we must amend the definitions of conjunction and disjunction. Some of the necessary modifications follow:

$$\left.\begin{array}{l} dis(\{<t1,t2>\},[t2]) = (t1,t2) \\ dis(\{<t1,t2>\},<t2,\ldots,t2>) = (t1,t2) \end{array}\right\} \text{ if } con(t1,ti) = ti$$

$$dis((t1,t2),[t2]) = dis((t1,t2),\{<t1,t2>\})$$

$$= dis((t1,t2),<t2,\ldots,t2>) = (t1,t2)$$

$$dis((t1,t2),\{[t2]\}) = (t1,dis(t2,[t2]))$$

$$dis((t1,t2),\{<t3,t4>\}) = (dis(t1,t3),dis(t2,t4))$$

$$dis((t1,t2),\{<t2,t2,t2>\}) = (t1,dis(t2,<t2,t2>))$$

$$con((t1,t2),\{<t1,t2>\}) = \{<t1,t2>\}$$

$$con((t1,t2),[t2]) = [t2]$$

$$con((t1,t2),<t2,\ldots,t2>) = <t1,\ldots,t2>$$

$$con((t1,t2),(t3,t4)) = (con(t1,t3),con(t2,t4))$$

$$\text{if } con(t2,t4) \neq tz$$

$$con((t1,t2),(t3,t4)) = tn|tt \text{ if } con(t2,t4) = tz$$

$$etc.$$

Even with the improvement just suggested, a subroutine parameter which is recognized as a mapping will be classified by our typefinder as of type (tg,tg). The typefinder has no way of determining the domain and range of a function from its uses, since even if the function is always indexed by an object of given type, the function might still record values for other indices of differing type. We can only

expect to determine the types of the domain and range of a map by applying the first method of type determination at the time that the map is passed as a parameter. Clearly, this requires that we develop some method of cross-subroutine optimization.

3. Applications of the Typefinder.

In the introduction of Chapter I we mentioned some of the applications which a type determination algorithm for compilers of very high level languages might be expected to have. In this section we describe some of these applications in more detail, and present more optimizations which can be realized once a typefinder is available.

A primary application of the typefinder is to enable a compiler to insert direct in-line code to perform a certain operation *op*. Without a knowledge of the types of the objects on which *op* is to be performed, one will prefer to call a run time routine which must dynamically determine the types of its operands and then call yet another run time routine to actually perform *op*. When types are known at compile time, we may even be able to eliminate the type-tags which objects must otherwise carry. When this is possible, objects such as integers can be kept in their actual machine representation m rather than in some more complex form l. We then avoid the packing and unpacking operations which must be done to convert from m to l and vice versa.

Although SETL allows long integers; that is, integers which cannot be contained within a single machine word, most SETL programs do not utilize this feature. The programmer who uses only short integers should not be penalized by the existence of language features which he does not use. This suggests that SETL should provide two modes of compilation. In the first mode, all integers identified by the typefinder are assumed to be short, allowing them to be represented directly in machine format. In the second mode, long integers are provided and long integer arithmetic is performed by calls on runtime subroutines. Note that even if the second mode is used, a savings can be realized by identifying objects as integers since only a single bit *b* need be kept at runtime to identify the integer as being short or long. This bit need not be kept packed in the same word as the integer but can be kept together with other such bits at a specific position of a run time bitstring. For integer operations one can then compile in-line code to test *b* and to execute a subroutine call if *b* is set but to execute in-line instructions otherwise. This will avoid expensive packing and unpacking operations. Note also that since this strategy allows removal of the type field from run time integers, longer integers can be considered "short" than would otherwise be possible. Thus we attain faster execution speed on a larger set of integers. Long/nolong mode selection can be done by the SETL user via a special declaration.

Similar optimizations can be performed in connection with SETL real numbers which are ordinarily represented by a root word which points to a two word block. If an object is identified as a real, it can be represented directly as a floating-point number in machine format which requires only one-third of the space which would othewise be needed. Tuples t of reals can then be represented as arrays of machine words containing real words rather than pointers. An invalid real bit pattern can be used to indicate the presence of an undefined atom within t. Sets of reals can be represented by hashtables which directly contain reals rather than pointers to reals. Of course, if machine format reals are used, there is no room in the word for a pointer to chain together set elements which has to the same value so that a rehashing system of resolving hash clashes rather than the chaining system which is currently used would be necessary.

Another optimization of the same sort is in the manipulation of logical values. As currently specified, logicals are represented by generalized bit strings of length 1. Thus each true/false value takes up an entire word complete with type indicator and possible chaining pointer. However, in most programs the only occurrences of bit strings are as logical values. If analysis shows an object to be a bit string of length 1, only a single bit need be kept to represent it. To save space, all such

logical bits can be packed in a single bit string with a specific position reserved for a specific logical value. For faster execution, it may be preferable to use a full word to store each logical value but that word can be kept in machine logical form without extra markers. To determine whether all bit strings in a program were logicals, we might require a programmer to declare whether or not he was using bit strings other than logicals, and compile his program accordingly. Alternatively, the type calculus can be extended to include two types, *tl* and *tb*, where *tl* represents "a logical value (i.e. a single bit)" and *tb* represents "a bit string of length possibly not equal to one." The type determination process itself would then determine whether or not a bit string is a logical.

Another advantage which may be gained from compile time type determination is that storage may be allocated at load time rather than dynamically during execution since storage needs for identifiers whose types have been determined are known at compile time. Further, the type determination process will identify type errors in a program, thus significantly assisting in program checkout.

Another point which may be made is that programs on which the typefinder is successful are "better" programs than those on which it fails, in the sense of being both more transparent to understand and less error prone. Thus, in some sense, the typefinder can serve as a guide to program

style and quality. Those cases where a "good" (i.e. typefinder-analyzable) program cannot be constructed are signals of weaknesses in a particular language and are areas where new language features might be needed.

Appendix: Tables for the *calc* and *backtype* functions.

In this appendix, we present the SETL code for the functions *calc* and *backtype* referred to in Chapter III. In these functions we use the routine *nstchk* whenever a type symbol is made a component of a set or tuple type symbol. *nstchk* accepts a type symbol t and returns the type symbol t' which is the same as t but with a nesting level limit of 2. Thus, when t' is inserted into a set or tuple type the resulting nesting level limit will be 3.

```
definef nstchk(x); t = x; /* to avoid changing x */
     if isel(t) then return t;;
     (3 ≤ ∀i ≤ #t) if n isel(t(i)) then
          /* t(i) is a gross type */
          ti = t(i);
            (3≤∀j ≤ #ti) /* search through each component
                                        of t(i) */
               if n isel(ti(j)) then
                    /* nesting level of 3 */
                    ti(j) = tg;  t(i) = ti;
               end if;
            end ∀j;
          end if;
     end ∀i;  return t;
end nstchk;
```

We now define the set *eltypes* of all non-erroneous elementary types; this set will be global to all subsequent routines.

```
eltypes = {ti,tb,tc,tt,tn,tg,ta,tu};
```

We also define a macro *istype* which takes two arguments and returns a logical result. *istype(t1,t2)* is true if and only if *t1* ≥ *t2* in our type calculus.

```
+* istype(t1,t2) = con(t1,t2) eq t2 **
```

We also define a set of macros which represent specific elementary alternations as follows.

```
+* tbcu = tb dis tc dis tu **
+* tin = ti dis tu **
+* tic = ti dis tc **
+* tbct = tb dis tc dis tt **
+* tibc = ti dis tb dis tc **
```

We also define an auxiliary function *nontu* which is given a type symbol *t* and returns a type symbol which is the same as *t* but does not include *tu* as an alternand. We assume that *tu* is represented by the bit string 00000001b.

```
+* nontuconst = 11111110b **
definef nontu(x);
    t = x; if t eq tg then return tg;;
    if isel(t) then return t and nontuconst;
    t(2) = t(2) and nontuconst; return t;
end nontu;
```

We are now in a position to give the code for the function *calc*. This routine calls on the routines *elun*, *cmun*, *elelbin*, *cmelbin*, *cmcmbin* and *elcmbin* which will be defined later.

```
definef calc(op,types);
    /* given an operator, op (see Ch. III, section 1 for
       the macros which represent operations), and a
       tuple of input types, types, calc returns the type
       which results (in our type calculus) when op is
       applied to types */
  initially
    opmap = /* a mapping which sends each operator either
         to the type of its output or to a label where
         that type is determined */
         {/* the following operators return integer results */
         <odv,ti>,<oabs,ti>,<omxm,ti>,<omnm,ti>,<osiz,ti>,
         /* the following return bitstrings */
         <oeq,tb>,<one,tb>,<ole,tb>,<ogt,tb>,<oge,tb>,
    <oelm,tb>,<olt,tb>,<onot,tb>,<oor,tb>,<oand,tb>,
    <oinc,tb>,
    /* the following returns tg */  <ord,tg>,
    /* the following are unary operators whose resultant
       type depends on the type of an input argument.
       unary is a label */
```

```
    <ohd,unary>,<otl,unary>,<oarb,unary>,<opw,unary>,
    <ooct,unary>,<odec,unary>,<oass,unary>,
    /* the following are binary */
    <oad,binary>,<osb,binary>,<oml,binary>,<orm,binary>,
    <owth,binary>,<olss,binary>,<olsf,binary>,<oof,binary>,
    <oofa,binary>,<onpw,binary>,
    /* the substring/subtuple operation is ternary */
    <ondx,ternary>,
    /* the following have an indeterminate number of
                                              arguments */
    <oset,setlab>,<otpl,tuplab>,<odxs,indexass>};
  end initially;
    ntypes = /* length of types tuple; i.e. number of operands*/
             #types;
    tl = /* type of first operand */ hd types;
    opmop = opmap(op);
    if type opmap eq bits then
         /*the operator always returns a specific type*/
                        return opmop;;
    /* else */
    go to /* the label at which the resultant type will be
             computed from the input types */
         opmop;
unary: /* single operand whose type is t1 */
    if isel(tl) then /* t1 is elementary. code for the
         routines elun and cmun will be given below */
                        return elun(op,tl);;
```

```
     /* t1 is a grosstype. elun will return the type result
        for the elementary alternand, and cmun for the
        nonelementary alternand */
     return dis(cmun(op,t1), elun(op,t1(2)));
binary:  /* case of a binary operator */
     t2 = types(2); /* type of the second operand */
     if isel(t1) then
          if isel(t2) then  /* .2 elementary operands */
               return elelbin(op,t1,t2);;
          /* code for elelbin, cmelbin, cmcmbin, and elcmbin
                          will be given below */
          /* first operand elementary, second not */
          return dis(elcmbin(op,t1,t2), elelbin(op,t1,t2(2)));
     end if;
     if isel(t2) then /* first operand nonelementary,
                          second elementary */
          retur1 dis(cmelbin(op,t1,t2), elelbin(op,t1(2),t2));;
     /* both operands non-elementary */
     return cmcmbin(op,t1,t2) dis elcmbin(op,t1(2),t2)
          dis cmelbin(op,t1,t2(2)) dis elelbin(op,t1(2),t2(2));
ternary: /* the only ternary operator is ondx */
     e11 = /* elementary alternand of indexed quantity */
          if isel(t1) then t1 else t1(2);
     /* check to make sure the second and third operands
                          are integers */
     if n istype(types(2),ti) or n istype(types(3),ti)
                          then return tz;;
```

```
    ternmap = /* a map from elementary input types to
                 corresponding types under ondx */
         {<tb,tb>,<tc,tc>,<tt,tu>,<tg,tbcu,tg>};
    x = /* tentative result type */ tz;
    (∀t ∈ eltypes)ttt = ternmap(t);
         if tt ne Ω and istype(ell,t)
                              then x = dis(x,ttt);;
    end ∀;
    if isunt(tl) then /* t1 is an unknown length tuple
                         being indexed */
                    x = dis(x,<unt,tu,tl(3)>);;
    if isknt(tl) then /* t1 is a known length tuple */
         x = dis(x,<unt,tu,[dis:3 ≤ i ≤ #tl] tl(i)>);;
    return x;
setlab: /* the operator is oset, the set former */
    /* check for tz among the components */
    if ∃t(i) ∈ types | nontu(t) eq tz then return tz;;
    return <set,tz,[dis: 1 ≤ i ≤ ntypes] nstchk(nontu(types(i)))>;
tuplab: /* the operator is otpl, the tuple former */
    /* check for tz among the components */
    if ∃t(i) ∈ types|t eq tz then return tz;;
    return <knt,tz> + [+: 1≤i≤ntypes]<nstchk(types(i))>;
indexass: /* the operator is odxs, indexed assignment of the
    form f(a,b,c) = y which is represented by
    <f,odxs,a,b,c,y,f> and where types =
    <type(a),type(b),type(c),type(y),type(f)> */
```

```
x = tz; /* initial approximation of result type */
ftype = /* type of object being indexed */
     types(ntypes);
if ntypes eq 3 and istype(tl,ti) then
     /* indexed assignment to a possible tuple,
        bitstring  or character string */
     if ftype eq tg then return tg;;
     if istype(ftype,tb) then x = dis(x,tb);;
     if istype(ftype,tc) then x = dis(x,tc);;
     if istype(ftype,tt) then /* indexed assignment
                                    to null tuple */
       /* check whether assigned value is Ω */
          asstyp = types(2);
          if istype(asstyp,tu) then x = dis(x,tt);;
       /* check whether assigned value can be
                        anything except Ω */
          if asstyp ne tu then
             x=dis(x,<unt,tz,dis(asstyp,tu)>);;
end if;
if isunt(ftype) or isunt(ftype) then
     x=dis(x,<unt,tz,dis(types(2),[dis:3≤i≤#ftype]
                                    types(i))>);;
end if;
/* we consider the possibility that ftype is a set */
if istype(types(ntypes-1),tu) then
     /* we must remove a tuple from ftype */
```

```
            t2 = calc(otpl,types(1:ntypes-2)+<tg>);
            x = dis(x,calc(olss,<ftype,t2>);
        end if;
        if types(ntypes-1) ne tu then /* add a tuple to ftype */
            types(ntypes-1) = nontu(types(ntypes-1));
            t2 = calc(otpl,types(1:ntypes-1);
            x = dis(x,calc(owth,<ftype,t2>);
        end if;
        return x;
end calc;
```

We now present the function *elun* which, given a unary operator *op* and an elementary alternation *t*, will return the resulting type symbol.

```
definef elun(op,t);
  initially
        elunmap = /* a map from operators and elementary input
                     types to result types */
            {<ooct,tc,ti>,<odec,tc,ti>,<ooct,ti,tc>,<odec,ti,tc>,
            <ooct,tg,tic>,<odec,tg,tic>,<oarb,tn,tu>,
            <oarb,tg,tg>,<opw,tn,<set,tz,tn>>,
            <opw,tg,<set,tz,<set,tn,tg>>>,<ohd,tt,tu>,
            <ohd,tg,tg>,<otl,tt,tu>,<otl,tg,<unt,tt,tg>>};
  end initially;
        if op eq oass /* assignment operation */ then return t;;
```

```
      x = tz; /* initial output type */
      (∀ttt∈ eltypes | istype(t,ttt))
           elres = elunmap(op,ttt);
           x = dis(x,elres orm tz);
      end ∀;
      return x;
end elun;
```

We now present code for the function *cmun*, which is given a unary operator *op* and a grosstype *t* and returns the type which results when *op* is applied to the non-elementary alternand of *t*.

```
definef cmun(op,t);
  initially
      cmunmap = /* a map from operators and grosstype classes
           to labels at which result types are computed */
           {<ohd,unt,hdtpl>,<ohd,knt,hdtpl>,<otl,unt,tlunt>,
           <otl,knt,tlknt>,<oarb,set,hdtpl>,<opw,set,pwset>};
  end initially;
      if op eq oass /* assignment operation */ then return t;;
      cmres = cmunmap (op, grostyp(t));
      if cmres eq Ω then return tz;;
      go to cmres;
hdtpl: return t(3);
tlunt: return <unt,tt,t(3)>;
```

```
tlknt: if #t eq 3 then return tt;;
     return <knt,tz> + t(4:#t);
pwset:  tl = t;  tl(2) = tz;
     return <set,tz,nstchk(tl)>;
end cmun;
```

The routine *elelbin* is called with an operator and two elementary alternations and returns the resulting type. The code follows:

```
definef elelbin(op,arg1,arg2);
  initially
     /* we first set up the necessary tables */
     intgad = /* an integer is being added */
          {<ti,ti>,<tg,ti>};
     bitsad = /* a bitstring is being "added" */
          {<tb,tb>,<tg,tb>};
     charad = /* a character string is being "added" */
          {<tc,tc>,<tg,tc>};
     nsetad = /* the null set is being "added" */
          {<tn,tn>,<tg,<set,tn,tg>>};
     ntplad = /* the null tuple is being "added" */
          {<tt,tt>,<tg,<unt,tt,tg>>};
     gnrlad = /* a general type is being "added" */
          {<ti,ti>,<tb,tb>,<tc,tc>,<tn,<set,tn,tg>>,
          <tt,<unt,tt,tg>>,<tg,tg>};
```

```
addmap = /* map for the plus operator */
     {<ti,intgad>,<tb,bitsad>,<tc,charad>,
     <tn,nsetad>,<tt,ntplad>,<tg,gnrlad>};
nsetsb = /* something is "subtracted" from the null set */
     {<tn,tn>,<tg,tn>};
gnrlsb = /* something is being "subtracted" from tg */
     {<ti,ti>,<tn,<set,tn,tg>>,<tg,<set,tin,tg>>};
submap = /* map for the minus operator */
     {<ti,intgad>,<tn,nsetsb>,<tg,gnrlsb>};
intgml =m/* an integer is being "multiplied"*/
     {<ti,ti>,<tb,tb>,<tc,tc>,<tg,tibc>};
nsetml = /* the null set is being "multiplied" */
     {<tn,tn>,<tg,tn>};
gnrlml = /*  tg is being "multiplied" */
     {<ti,ti>,<tb,tb>,<tc,tc>,<tn,tn>,<tg,<set,tibcn,tg>>};
mulmap = /* map for the "times" operator */
     {<ti,intgml>,<tn,nsetml>,<tg,gnrlml>};
gnrlrm = /* tg is used in a remainder operation */
     {<ti,ti>,<tn,<set,tn,tg>>,<tg,<set,tin,tg>>};
remmap = /* map for the // operator */
     {<ti,intgad>,<tn,nsetad>,<tg,gnrlrm>};
bitsof = /* indexing a bitstring */
     {<ti,tb>,<tg,tb>};
charof = /* indexing a character string */
     {<ti,tc>,<tg,tc>};
```

```
    ntplof = /* indexing the null tuple */
         {<ti,tu>,<tg,tu>};
    oofmap = /* map for the oof operation */
         {<tb,bitsof>,<tc,charof>,<tt,ntplof>,<tn,tu>,
                        <tg,tg>};
    intnpw = /* integer used in an onpw operation */
         {<tn,<set,tn,tn>,<tg,<set,tn,tg>>};
    npowmap = /* map for onpw operation */
         {<ti,intnpw>};
    lessmap = /* map for olss operation */
         {<tn,tn>,<tg,<set,tn,tg>>}
    binbinmap =   {<oad,addmap>,<osb,submap>,<oml,mulmap>,
         <orm,remmap>,<olss,lessmap>,<olsf,lessmap>,
         <onpw,npowmap>,<oof,oofmap>,<∞fa,lessmap>};
end initially;
    if op eq owth then /* special case for owth operator */
         if istype(arg1,tg) then return <set,tz,tg>;;
         if istype(arg1,tn) then a2 = nontu(arg2);
              if a2 ne tz then return <set,tz,a2>;;;
    end if;
    x = tz;  opermap = binbinmap(op); if opermap eq Ω then
                                        return x;;
    (∀t  ∈ eltypes|istype(arg1,t))
         (∀ttt∈eltypes|istype(arg2,ttt))
              opt = opermap(t);
              if type opt ne set and opt ne Ω
                   then x = dis(x,opt);
```

```
                    else if opt ne Ω then
                            optt = opt(ttt);
                            if optt ne Ω then x=dis(x,optt);;
                        end if;
                end if;
            end ∀ttt;
        end ∀t;
        return x;
end elelbin;
```

The routine *elcmbin* is called with an operator, an elementary alternation *arg1* and a grosstype *arg2* and returns the type which results when the binary operator is applied to *arg1* and the non-elementary alternand of *arg2*.

```
definef elcmbin(op,arg1,arg2);
    /* we first set up a set of mappings for each operation
       from an elementary type and a grosstype to a
       resulting type  */
    a2 = nontu(arg2);
    setarg2 = <set,tz,nstchk(a2)>; stng=<set,tn,tg>;
                        stzg = <set,tz,tg>;
    if a2 eq tz then setarg2 = tz;;
    elcmad = /* plus operation */
        {<tn,set,arg2>,<tg,set,stzg>,<tt,unt,arg2>,
        <tg,unt,<unt,tz,tg>>,<tt,knt,arg2>,<tg,knt,<unt,tz,tg>>};
```

```
elcmsb = /* minus operation */
     {<tn,set,tn>,<tg,st,stng>};
elcmofa = /* oofa operation */
     {<tn,set,tn>,<tn,unt,tn>,<tn,knt,tn>,<tg,unt,stng>,
     <tg,set,stng>,<tg,knt, stng>};
elcmlss = /* less operation */
     {<tn,set,tn>,<tn,unt,tn>,<tn,knt,tn>,<tg,set,tg>,
     <tg,unt,tg>,<tg,knt,tg>};
elcmrm = /* // operation */
     {<tn,set,arg2>,<tg,set,stng>};
elcmwth = /* with operation */
     {<tn,set,setarg2>,<tn,knt,setarg2>,<tn,unt,setarg2>,
     <tg,set,stzg>,<tg,unt,stzg>,<tg,knt,stzg>};
elcmof = /* oof operation */
     {<tn,set,tn>,<tn,unt,tu>,<tn,knt,tu>,<tg,set,tg>,
     <tg,knt,tg>,<tg,unt,tg>};
elcmnpw = /* npow operation */
     {<ti,set,setarg2>,<tg,set,setarg2>};
/* we now establish a mapping from the binary operators
   into the correct mapping */
elcmmap = {<oad,elcmad>,<osb,elcmsb>,<oml,elcmsb>,
     <orm,elcmrm>,<owth,elcmwth>,<olss,elcmlss>,
     <olsf,elcmlss>,<oof,elcmof>,<oofa,elcmofa>,
     <onpw,elcmnpw>};
x = tz;
```

```
    opermap = elcmmap(op); if opermap eq Ω then return x;;
    (∀t ∈ eltypes|istype(arg1,t)) opt=opermap(t,arg2(1));
        if opt ne Ω then x = dis(x,opt);;
    end ∀;
    return x;
end elcmbin;
```

The routine *cmelbin* is called with an operator, a grosstype *arg1* and an elementary alternation *arg2* and returns the type which results when the binary operator is applied to the non-elementary alternand of *arg1* and to *arg2*.

```
definef cmelbin(op,arg1,arg2);
  initially
    /* set up mappings from the operator and the grosstype
       of arg1 to a label where the result type is computed*/
    cmelmap = {<osb,set,cmelsb>,<owth,set,cmelwth>,
        <olss,set,cmelss>,<olsf,set,cmelss>,
        <oof,set,cmofst>,<oof,unt,cmofunt>,
        <oof,knt,cmofknt>,<oofa,st,cmofast>};
  end initially
    if op∈{oad,osb,orm} then /* commutative operator */
        return elcmbin(op,arg2,arg1);;
    label = cmelmap(op,arg1(1));
    if label eq Ω then return tz;;
    go to label;
  cmelsb: return dis(if istype(arg2,tn) then targ1 else tz,
                  if arg2 eq tg then <set,tn,arg1(3)>else tz);
```

```
  cmelwth: ret = nontu(dis(arg2,argl(3)));
                          if ret eq tz then return tz;;
     return<set,tz,ret>;
  cmelss: return<set,tn,argl(3)>;
  cmofst: return fnappl(targl,targ2);
     /* code for fnappl will be given below */
  cmofunt: return if istype(arg2,ti) then dis(tu,argl(3)) else tz;
  cmofknt: return if istype(arg2,ti) then dis(tu,
                          [dis: 3≤i≤#argl] argl(i));
  cmofast: return<set,tn,nstchk(fnappl(a,b))>;
end cmelbin;
```

The routine *cmcmbin* is called with a binary operator *op*, and two grosstypes, *arg1* and *arg2*. It returns the type which results when *op* is applied to objects of the types given by the non-elementary alternands of *arg1* and *arg2*.

```
definef cmcmbin(op,argl,arg2);
  initially
     /* set up mappings for each operator from the two gross-
        types into a label where the result type will
        be computed */
     cmcmad = /* the plus operation */
          {<set,set,ssad>,<unt,unt,ssad>,<knt,knt,kkad>,
          <unt,knt,ukad>,<knt,unt,kuad>};
     cmcmsb = /* the minus operation */
         {<set,set,sssb>};
```

```
   cmcmml = /* the times operation */ {<set,set,ssml>};
   cmcmrm = /* the // operation */ {<set,set,ssrm>};
   cmcmlss = /* the less operation */
        {<set,set,sssb>,<set,unt,sssb>,<set,knt,sssb >}
   cmcmwth = /* the with operation */
        {<set,set,sswth>,<set,unt,sswth>,<set,knt,sswth>};
   cmcmof = /* the oof operation */
        {<set,set,ssof>,<set,unt,ssof>,<set,knt, ssof>};
   cmcmofa = /* the oofa operation */
        {<set,set,sofa>};
   /* we now set up the mapping which takes each operator
      into its proper map */
   cmcmmap = {<oad,cmcmad>,<osb,cmcmsb>,<oml,cmcmml>,
        <orm,cmcmrm>,<owth,cmcmwth>,
        <olss,cmcmlss>,<olsf,cmcmlss>,
        <oof,cmcmof>,<oofa,cmcmofa>};
end initially;
   opermap = cmcmmap(op);
   if opermap eq Ω then return tz;;
   opt = opermap(arg1(1),arg2(1));
   if opt eq Ω then return tz;;
   go to opt;
ssad: return <arg1(1),tz,dis(arg1(3),arg2(3))>;
kkad: return<knt,tz> + tℓ tℓ arg1 + tℓ tℓ arg2;
knad: <arg2,arg1> = <arg1,arg2>;
ukad:return <unt,tz,dis(arg1(3),[dis:3≤i≤#arg2]arg2(i))>;
```

```
  sssb: return <set,tn,argl(3)>;
  ssml: return <set,tn,con(argl(3),arg2(3))>;
  sswth: ret = nontu(dis(argl(3),nstchk(arg2)));
         if ret eq tz then return tz;;
     return <set,tz,ret>;
  ssof: return fnappl(argl,arg2);
  sofa: return <set,tn,nstchk(fnappl(argl,arg2))>;
  ssrm: return <set,tn,dis(argl(3),arg2(3))>;
end cmcmbin;
```

We now present code for the routine *fnappl* which takes two arguments *a* and *b*. *a* is a set type and *b* an elementary alternation. *fnappl(a,b)* returns the type which results when an object of type *b* is used as a functional index to an object of the type of the non-elementary alternand of *a*.

```
definef fnappl(a,b);
     compa = /* the components of a */ a(3);
     if isel(compa) then /* elementary components */
          return if compa eq tg then tg else tu;;
     if isset(compa) then /* set components */ return tu;;
     if #compa le 3 then return tu;;
     if con(compa(3),b) eq tq then return tu;;
     if isel(b) then /* b is elementary */
          if isknt(compa) then
               if #compa eq 4 then return dis(tu,compa(4));;
```

```
                return< knt,tu> + compa(4:#compa);
            end if;
            /* b is elementary and a is of grosstype unt */
            return <unt,compa(3),compa(3)>;
        end if;
        /* b is a set or tuple */
        x = tu;
        if isknt(compa) then
            if #compa eq 4 then x = dis(x,compa(4));
                else x = dis(x,<knt,tz>+compa(4:#compa));;
        end if;
        /* compa is a tuple of unknown length */
        x = dis(x,<unt,compa(3),compa(3)>);
        return dis(x,fnappl(a,b(2)));
end fnappl;
```

This concludes the set of routines which are necessary for computation of the *calc* function and the forward method of type determination in SETL.

We now present the routine *backtype* which is used in the second method of type determination (see Chapter III, section 5). The routine accepts a use *u* and returns the type information concerning *u* which can be deduced from the operator to which *u* is an argument and the type of object which this operator produces. Only information which cannot be deduced from the forward method is returned.

```
definef backtype(u);
     tup = /* the defining tuple in which u appears */
          deftup(d);
     op = /* the operator applied to u */ tup(2);
     pos = /* the position of u as an input use */ u(4);
     if op ∈ /* the set of operations which apply only
                                        to integers */
          {oabs,odv,omxm,omnm,olt,ole,ogt,oge} or
          (op ∈ {ondx,onpw} and pos gt 1 then return ti;;
     if op ∈ /* set of operators applying to general types */
          {oeq,one,oof,oset,otpl,odxs,osiz} or
          (op∈{owth,olss,olsf,oofa} and pos eq 2) or
          (op eq oelm and pos eq 1) then return tg;;
     if op ∈ /* set of operators applying only to sets */
          {opw,oarb,oinc,owth,olss,olsf,onpw,oelm,oofa}
               then return <set,tn,tg>;;
     if op ∈ /* set of operators applying only to bitstrings*/
          {onot,oand,oor} then return tb;;
     if op ∈ /* set of operators applying only to tuples */
          {ohd,otl} then return <unt,tt,tg>;;
     if op eq ondx then return <unt,tbct,tg>;;
     dtype = /* the type of the defintion to which u is input*/
          type(<tup(1),u(2),u(3)>);  x=tz;
     if op ∈ {ooct,odec} then
          if istype(dtype,ti) then x = dis(x,tc);;
          if istype(dtype,tc) then x = dis(x,ti);;
          return x;
     end if;
```

```
    /* at this point, op is either oad, oml, osb, oass or orm */
    if op eq oad and isknt(dtype) then
         return <unt,dtype(2), [dis:3≤i≤#dtype]dtype(i)>;;
    if op ne oml then return dtype;;
    /* at this point, op is oml */
    if pos eq 2 then
         if isset(dtype) then x = dis(x,<set,con(tibc,dtype(2)),
                                                        tg>);;
         if istype(dtype,tn) then x=dis(x,<set,tn,tg>);;
         if isel(dtype) then x=dis(x,con(dtype,tibc));;
    end if;
    if isel(dtype) then del=dtype;
                   else del=dtype(2);;
    if istype(del,tibc) then x = dis(x,ti);;
    if istype(del,tn) then x = dis(x,<set,tn,tg>);;
    if isset(dtype) then x = dis(x,<set,tz,tg>);;
    return x;
end backtype;
```

Bibliography

[1] Allen, F. E., Program Optimization, Annual Review in Automatic Programming, Pergamon Press, New York (1969), Volume 5, pp. 239-307.

[2] Allen, F. E., Control Flow Analysis, SIGPLAN Notices, Vol. 5, No. 7 (1970), pp. 1-19.

[3] Allen, F. E., A Basis for Program Optimization, Proceedings of the IFIPS Congress 1971, North-Holland Publishing Company, Amsterdam, Holland.

[4] Allen, F. E., Interprocedural Data Flow Analysis, IBM Research Report RC 4633, IBM Research Center, Yorktown Heights, New York (1973).

[5] Bauer, A. and Saal, H., Does APL Really Need Run-time Checking? Software Practice and Experience, Vol. 4, No. 2 (April-June 1974).

[6] Brown, S. and Harrison, M., The BALM Programming Language, Courant Institute of Mathematical Sciences, New York University (1973).

[7] Kennedy, K., An algorithm for live-dead analysis including node-splitting for irreducible program graphs, SETL Newsletter No. 38, Courant Inst. Math. Sci., New York Univ. (1972).

[8] Kildall, G. A., A unified approach to global program optimization, ACM Symposium on Principles of Programming Languages, Assoc. for Computing Machinery, New York, N. Y. (1973).

[9] Ledgard, H. F., A Model for Type Checking -- With an Application to ALGOL 60. Communications of the ACM Vol. 15, No. 11 (November 1972).

[10] Owens, P. and Kennedy, K., Initial Description of an Algorithm for Use-Definition Chaining in Optimization, SETL Newsletter No. 37, Courant Inst. Math. Sci., New York Univ. (1971).

[11] Schwartz, J. T., More detailed suggestions concerning "data strategy" elaborations for SETL, SETL Newsletter No. 39, Courant Inst. Math. Sci., New York Univ. (1971).

[12] Schwartz, J. T., Deducing the logical structure of objects occurring in SETL programs, SETL Newsletter No. 71, Courant Inst. Math. Sci., New York Univ. (1972).

[13] Schwartz, J. T., On Programming: An interium report on the SETL project, Courant Inst. Math. Sci., New York Univ. (1973a,b). Installment 1: Generalities. Installment 2: The SETL language, and examples of its use.

[14] Schwartz, J. T., Automatic and Semiautomatic Optimization of SETL, SIGPLAN Notices, Vol. 9, No. 4, (April 1974 a), pp. 43 ff.

[15] Schwartz, J. T., Some optimizations using type information, SETL Newsletter No. 132, Courant Inst. Math. Sci., New York Univ. (1974 b).

[16] Schwartz, J. T., Inter-Procedural Optimization, SETL Newsletter No. 134, Courant Inst. Math. Sci., New York Univ. (1974 c).

[17] Sintzoff, M., Calculating Properties of Programs by Valuations on Specific Models, SIGPLAN Notices, Vol. 7, No. 1 (1972).

[18] Tenenbaum, A., Revised and Extended Algorithms for Deducing the Types of Objects Occurring in SETL Programs. SETL Newsletter No. 118, Courant Inst. Math. Sci., New York Univ. (1973).

[19] Warren, H., SETL Compiled Code: Calls to SETL Procedures, SETI Newsletter No. 60, Courant Inst. Math. Sci., New York Univ. (1971).

N.Y.U. Courant Institute of Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012