A General Scheme for the Automatic Inference of Variable Types†
Extended Abstract

by

Marc A. Kaplan
&
Jeffrey D. Ullman
Princeton University

Summary

We present the best known algorithm for the determination of run-time types in a programming language requiring no type declarations. We demonstrate that it is superior to other published algorithms and that it is the best possible algorithm from among all those that use the same set of primitive operators.

I Introduction

In this paper we present a model of computation that is an abstraction of typeless programming languages such as APL, SETL and SNOBOL. Based on this model we present a general scheme for automatically inferring the types of the variables in a given program. Our type inference system is provably more powerful than the systems of Jones and Muchnick [J] and Tenenbaum [T].

In Section II we introduce a model for the treatment of type inference. A system of relationships that enables us to infer types is discussed in Section III. Then Section IV gives a formula that we believe is the best achievable without introducing wholly new concepts into the problem of type determination. Section V justifies our view by showing how to get the best possible result from a given set of primitive operators and starting solutions. In Section VI we show that our solution is at least as good as other proposed methods, and Section VII is an extended example that demonstrates our solution can be better than other known solutions.

---

†Work partially supported by NSF grant MCS-76-15255.

II A Model of Computation in a Programming Language

The basic building block of our programming language is the parallel assignment statement, whose most general form is Q:

$$(X_1,X_2,X_3,\ldots,X_k)\leftarrow(\oplus_{i_1}(Y_{11},Y_{12},\ldots,Y_{1d_1}),\ \oplus_{i_2}(Y_{21},Y_{22},\ldots,Y_{2d_2}),\ \ldots,\ \oplus_{i_k}(Y_{k1},Y_{k2},\ldots,Y_{kd_k}))$$

where

$X_1,X_2,\ldots,X_k$ are distinct variable names,
$\oplus_{i_1},\oplus_{i_2},\ldots,\oplus_{i_k}$ are operators of degrees $d_1,d_2,\ldots,d_k$, respectively, and
the Y_{jm}'s are variable names

The simple assignment $X\leftarrow Y$ is included by writing it as $X\leftarrow id(Y)$, where id is the identity operator. Operators may be of degree 0; that is they may take no arguments and yield constant values.

The intent is that variable names are bound to (associated with) values from (members of) a universe of values V. Each operator $\oplus_j$ with degree d_j corresponds to a function $\Theta_j:V^{d_j}\rightarrow V$.

Now the meaning of an assignment statement should be clear. Let the variables Y_{jm} be bound to values v_{jm} before assignment Q is executed. Then after Q is executed variables $X_1,\ldots,X_k$ become bound

to the values $\psi_{j_1}(v_{11},\ldots,v_{1d_1}),\ldots$ $\psi_{j_k}(v_{k1},\ldots,v_{kd_k})$, respectively. Each x_i loses any value it may have been bound to before the execution of Q. All other program variables Y_{jm} retain their values.

A program is a directed graph, each of whose nodes is labeled by an instance of an assignment statement, subject to the following restrictions:

1. There is a special node which we call the start/finish (SF) node.
2. Every node is reachable from the SF node.
3. The SF node is reachable from every node.
4. The SF node is labeled by an assignment statement in which every program variable appears on the left hand side and no variables appear on the right hand side.

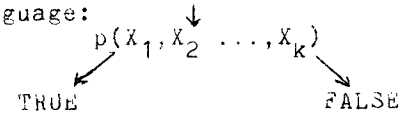
A program execution is a path through the program graph that begins at the SF node and ends at the SF node but does not contain any other instances of the SF node.

Correspondence of the model programming language to real programming languages

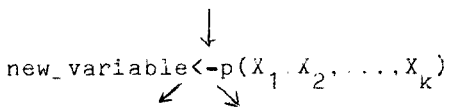
Executing the SF node corresponds to initializing all variables. Reaching the SF node corresponds to program termination. If we like, we may postulate the existence of a special value, undefined $\in$ V, and a constant function, μ, which yields this value. Then a typical program might have its SF node labeled with the statement $(Z_1,Z_2,\ldots,Z_k)\leftarrow(\mu,\mu,\ldots,\mu)$.

We can model the input and output statements of a real programming language with special operators. Decision statements can be modeled as follows:

real language:



model PL:



Admittedly, we only model the fact that the predicate, p, is evaluated in a model program either edge may be taken after p is evaluated, regardless of p's outcome. Thus there may exist execution paths in the model of a real program which are not execution paths in the real program. But it is true that every possible real execution path is a possible path through the model program Therefore, any statement that holds true for all executions of a model program P, must also hold true for every possible execution of the program that P models. If we take any reasonable semantics for predicates, the set of program executions is not effectively computable, so our over-estimate of the set of execution paths is a reasonable approach, and corresponds to the assumptions usually made in code optimization [A, G, H, Ka, Ki, Sc].

We shall assume that all programs that we analyze are correct, in the sense that the result of every operation will be well defined for all input values occurring during any execution, and that all executions reach the SF node after passing through a finite number of statements.

III The Type Determination Problem

If a compiler could predict the range of values that variables may take on during the execution of a program, it could utilize the information to produce simpler, more efficient code to carry out the operations indicated by the program. For example, if the statement I←J+K occurred somewhere in the text of an APL program, and the compiler could determine that both J and K would always be bound to scalar integer values for every execution of this statement, then the compiler could generate code that would use a simple machine instruction to add the values of J and K (rather than calling a generalized addition routine), and could allocate a single word for I in which to place the result (rather than binding I to the storage that the generalized addition routine would allocate dynamically). Moreover, if a global check showed that I never takes on any values other than those of scalar integers, then the compiler could allocate I at compile time (assuming no recursive uses of I). Bauer [B] shows that even a simple type determination scheme can result in substantial savings.

More generally, we would like to partition the universe of computable values, V, into "types. For the purposes of this paper, a type is any subset of V. For instance, types may correspond to different storage representations. Each of integer, real array[1:10], real set, heterogeneous array, char, char string, anything (a data structure capable of storing any value in the universe V) might be types. It is perfectly permissible for one type to contain values that are also contained in another type. It is also possible to incorporate the constant propagation problem into our scheme by

having types such as constant 3, constant string "abc", and so on.

In general, the compiler writer must decide on the set of types he is willing to consider. Here we shall only make the assumptions that:

1. The set of types, which we shall denote by $\mathbb{T}$, forms a lattice.‡ The meet (greatest lower bound) of two elements $s,t \in \mathbb{T}$ is denoted by $s \wedge t$. The join (least upper bound) of s and t is denoted by $s \vee t$. A partial ordering $\leq$ on $\mathbb{T}$ is given by the rule $y \geq x \iff x \leq y \iff x \wedge y = x \iff y \vee x = y$. We shall use $<$ for "$\leq$ but not $=$." There exists a least element $\underline{0}$ and a greatest element $\underline{1}$ such that for all $s \in \mathbb{T}$, $\underline{0} \leq s \leq \underline{1}$.

   The partial ordering $\leq$ should be interpreted as follows. "If $s \leq t$ then saying that a program variable X is bound to a value of type s is more precise than merely saying that X is bound to a value of type t."

2. A symbol or expression that represents an element of $\mathbb{T}$ will usually appear surrounded by brackets when it is important that the reader think of that expression as a subset of V as well as a member of $\mathbb{T}$. Bearing this in mind, we assume that the lattice join and meet are related to set union and intersection as follows:
   $[t \vee s] \supseteq [t] \cup [s]$ and
   $[t \wedge s] \supseteq [t] \cap [s]$.

   This should be interpreted as follows. "If program variable X is either bound to a value of type t or a value of type s, then it is surely bound to a value of type $t \vee s$, which must contain the set $[t] \cup [s]$. On the other hand, if we know that X is bound to a value whose type is described by both t and s, then X is surely bound to a value whose type is $t \wedge s$, which must contain the set $[t] \cap [s]$."

3. (Finite Chain Condition) All chains (sequences of elements related by $<$) in $\mathbb{T}$ are of finite length. Note that this is not so strong a condition as assuming that all chains have their length bounded by some constant. However, the finite chain condition does imply that $\mathbb{T}$ is a complete lattice.‡‡

‡ An introduction to lattice theory and its basic definitions may be found in reference [D].

‡‡ A lattice is complete if every chain of elements has both a least upper bound and a greatest lower bound.

To see that these assumptions are reasonable, let us consider how such a lattice of types may be constructed. First, we choose some "basic" subsets of V, such as the example types mentioned above. We may also include some "alternated types [T] such as real or character or integer or integer array in this initial collection of types. A natural partial ordering of these subsets is given by the subset relation, $\supseteq$. To insure the existence of greatest and least types we add V and the empty set to our collection of types. Next, we must extend our collection so that for each pair s and t of subsets of V that appear in our collection, there also appears both a unique largest subset that contains the intersection of s and t and a unique smallest subset that contains the union of s and t. This assures the existence of a greatest lower bound and a least upper bound for each pair in our collection. Our collection is now a lattice satisfying assumptions 1 and 2. If we were careful in all of our selections, then we should also satisfy assumption 3. In particular, if we started with only finitely many "basic" subsets, then our extension to a complete lattice will also be finite

Another way to think about the construction of a lattice of types, $\mathbb{T}$, is to view the process as the judicious selection of a subpartial order of the powerset of V that is coincidentally a lattice satisfying assumptions 1,2 and 3. Tenenbaum [T] gives a very detailed example of such a lattice of types for the programming language SETL.

Type Inference

Our goal is to automatically infer the type of values each variable may be bound to at each point in the program. This information will aid the compiler in deciding what code to generate for each statement.

We begin the task by examining the operators of our programming language. Suppose we know that at some point in a program, X_i is bound to a value of type t_i, for $1 \leq i \leq k$. Now consider the expression $\oplus(X_1, X_2, \ldots, X_k)$. We would like to speak about its type. $\oplus$ names a function, so consider the set $R = \{\oplus(v_1, \ldots, v_k) \mid v_i \in [t_i]\}$. R necessarily includes all of the values that $\oplus(X_1, X_2, \ldots, X_k)$ may have. Now let s be an element of $\mathbb{T}$ such that $[s] \supseteq R$. Then it is clear that the value of $\oplus(X_1, X_2, \ldots, X_k)$ must be of type s. Thus for each operator $\oplus$, we may define a function $T^0_\oplus : \mathbb{T}^k \to \mathbb{T}$ such that $T^0_\oplus$ is the least type s such that $[s]$

$\supseteq \{\oplus(v_1,\ldots,v_k) \mid v_i \in [t_i]\}$. ‡

A function f on a lattice is said to be monotonic if $f(x) \leq f(y)$ whenever $x \leq y$.

Lemma 1: $T^0_\oplus$ is monotonic.

Proof: Let $(q_1,\ldots,q_k) \geq (r_1,\ldots,r_k)$. Define $Q = \{\oplus(v_1,\ldots,v_k) \mid v_i \in q_i\}$ and $R = \{\oplus(v_1,\ldots,v_k) \mid v_i \in r_i\}$. If $[s] \supseteq Q$, then $[s] \supseteq R$. Therefore $[T^0_\oplus(q_1,\ldots,q_k)] \supseteq R$. Since $T^0_\oplus(r_1,\ldots,r_k) \supseteq R$ by the definition of $T^0_\oplus$ it follows that

$$[T^0_\oplus(q_1,\ldots,q_k) \bigwedge T^0_\oplus(r_1,\ldots,r_k)] \supseteq R$$

by our assumption that $[t \bigwedge s] \supseteq [t] \cap [s]$. Therefore

$$T^0_\oplus(q_1,\ldots,q_k) \bigwedge T^0_\oplus(r_1,\ldots,r_k) = T^0_\oplus(r_1,\ldots,r_k),$$

so $T^0_\oplus(q_1,\ldots,q_k) \geq T^0_\oplus(r_1,\ldots,r_k)$. □

We interpret the monotonicity of our type inference functions as being the condition that "the more precise the information we have concerning the type of an operator's inputs, the more precise we can be in making a statement about the operator's output."

We have considered the case where the types of an operator's inputs are known, and we have seen how we can deduce a valid assertion about the operator's result. Now suppose that we have some a priori information concerning the type of an operator's result and the types of some of its inputs. We may be able to capitalize on this information by using it to deduce something more about a particular input. For example consider the APL statement: A←B+1. Suppose we have knowledge that A will be an integer after the statement has been executed. Then by the nature of the '+' operator, it must be that B is bound to an integer value before the '+' operator is applied.

‡ Note that $\mathbf{T}^K$ is a lattice with partial order $\leq$ given by $(x_1,\ldots,x_k) \leq (y_1,\ldots,y_k)$ if and only if $x_i \leq y_i$, for $1 \leq i \leq K$, where $x_i, y_i \in \mathbf{T}$. $\bigwedge$ and $\bigvee$ are similarly extended; if $x,y \in \mathbf{T}^K$, then $x \bigwedge y$ is the greatest element $w \in \mathbf{T}^K$ and $x \bigvee v$ is the least element $u \in \mathbf{T}^K$, such that $w \leq x \leq u$ and $w \leq y \leq u$. Moreover, if $\mathbf{T}$ satisfies the finite chain condition, then so does $\mathbf{T}^K$.

As we constructed a function $T^0_\oplus$ for each operator $\oplus$, so we now construct additional functions $T^1_\oplus, T^2_\oplus, \ldots, T^K_\oplus$ for each operator as follows. (k is the degree of operator $\oplus$.) For each $j=1,2,\ldots,K$ construct a monotonic function $T^j_\oplus: \mathbf{T}^{K+1} \to \mathbf{T}$ such that $T^j_\oplus(t_0,t_1,t_2,\ldots,t_k)$ is the least element of $\{ s \in \mathbf{T} \mid [s] \supseteq \{v_j \mid v_j \in [t_j]$ and $\exists\, v_i \in [t_i]$, $i \neq j$ such that $v_0 = \oplus(v_1,\ldots,v_k)\}\}$. We may prove each $T^j_\oplus$ to be monotonic; the proof is similar to that of Lemma 1 and is omitted. ‡

Inference of types across statements

Now we would like to consider what type inferences can be made by examining a single assignment statement, Q, such as the one shown in Sect. 2. Type determination is unusual among data flow problems in that information may be propagated both forward and backward.

Forward inference: Let t be a map from the set z of program variables to the elements of $\mathbf{T}$ giving knowledge about the types of the values assigned to the program's variables before statement Q is executed. We wish to construct from t and Q a new map $f_Q(t)$ that describes what we know about the types of the values assigned to the program variables after statement Q is executed. In constructing f_Q we make use of our knowledge of the operators and the semantics of the assignment statement.

For each variable $Z \in z$ we define $[f_Q(t)](Z)$ to be

(1) $t(Z)$, if Z does not occur in statement Q.

(2) $T^0_{\oplus_{i_m}}(t(Y_{m1}), t(Y_{m2}), \ldots, t(Y_{md_m}))$, if Z occurs in the m^{th} position of the left hand side of Q.

‡ We expect that it will not always be practicable to construct these best type inference functions $T^j_\oplus$. In such cases any monotonic functions $T^j_\oplus$, which approximate the functions $T^j_\oplus$ may be used in the theory which follows. We say g approximates f if, for all x in the domain of f, $g(x) \geq f(x)$.

(3) $\bigwedge_{\text{all pairs } (m,j) \text{ such that } Z=Y_{mj}} T^{j}_{\oplus_{i_m}}(\underline{1}, t(Y_{m1}), t(Y_{m2}), \ldots, t(Y_{md_m}))$, otherwise.

Backward inference: Let t be a map from the set of program variables z to the elements of $\mathbb{T}$ giving knowledge about the types of the values assigned to the program's variables after statement Q is executed. We wish to construct a new map $b_Q(t)$ which will describe what we can deduce about what the types of the values of the program variables were before statement Q was executed. In constructing b_Q we again make use of our knowledge of the operators and the semantics of the assignment statement.

Define $[b_Q(t)](Z)$ to be

(1) $t(Z)$, if Z does not occur in statement Q.

(2) $\underline{1}$, if Z occurs only on the left hand side of Q.

(3) $\bigwedge_{\text{all pairs } (m,j) \text{ such that } Z=Y_{mj}} T^{j}_{\oplus_{i_m}}(t(X_m), t'(Y_{m1}), t'(Y_{m2}), \ldots, t'(Y_{md_m}))$,

otherwise.

where

$$t'(W) = \begin{cases} \underline{1}, & \text{if W appears on the left side of Q} \\ t(W), & \text{otherwise} \end{cases}$$

Notice that both $f_Q(t)$ and $b_Q(t)$ can be regarded as functions of t for a fixed statement Q. Indeed, f_Q and b_Q are monotonic functions that map $[z\to\mathbb{T}]$ into $[z\to\mathbb{T}]$[*], since they are just compositions of the monotonic functions $\{T^{j}_{\oplus_i}\}$ and $\bigwedge$.[‡]

[*] $[z\to\mathbb{T}]$ denotes the set of all mappings from the set z to the lattice $\mathbb{T}$. Note that $[z\to\mathbb{T}]$ is a lattice. If $f,g \in [z\to\mathbb{T}]$ then we define $f\le g$ whenever $f(Z)\le g(Z)$, for all $Z\in z$. Further, if z is a finite set of ℓ elements then $[z\to\mathbb{T}]$ is isomorphic to $\mathbb{T}^{\ell}$, whence the finite chain condition holds for $[z\to\mathbb{T}]$ whenever it holds for $\mathbb{T}$.

Program-wide type inference

Now consider a program P in our model language consisting of a directed graph of n nodes. For convenience let the SF node be node 1, and designate the other nodes $2,3,\ldots,n$.

Let the program variables be $z = \{Z_1, Z_2, \ldots, Z_\ell\}$. At any given point in the program we describe the types of the values which may be assigned to each of the variables by a mapping $t: z\to\mathbb{T}$. A mapping t safely describes the types of values of z if, for each i, $[t(Z_i)] \supseteq \{v\in V \mid v \text{ is possibly bound to } Z_i \text{ during any program execution}\}$. It is easy to see that if s safely describes z, then all mappings x such that $x(Z_i)\ge s(Z_i)$ for all $Z_i\in z$ safely describe z. A mapping t is more precise (better) than a safe mapping s if t is safe and $t\le s$.

For each node j of P we construct forward and backward inference functions f_j and b_j from $[z\to\mathbb{T}]$ to $[z\to\mathbb{T}]$, as outlined above for example node Q.

By construction we have:

1. If $t: z\to\mathbb{T}$ safely describes the types of the values assigned to the program variables before node j is executed, then $f_j(t)$ safely describes the types of the values assigned to the program variables after node j has been executed.

2. If $s: z\to\mathbb{T}$ safely describes the types of the values assigned after node j has been executed, then $b_j(s)$ safely describes the types of the values which were assigned to the program variables before node j was executed.

Now, couching our type inference problem in the notation we have developed above, our problem is to find mappings $x_1, x_2, x_3, \ldots, x_n$ from z to $\mathbb{T}$ so that for each x_j and for all executions of our program P and for all times in a given execution of P at which control reaches node j, x_j safely describes the program variables on entry to node j. We call such a set of mappings a safe solution to the type finding problem for program P under the lattice of types $\mathbb{T}$. It will often be convenient to write a proposed solution as one vector

$$x = (x_1, x_2, x_3, \ldots, x_n) \in [z\to\mathbb{T}]^n.$$

[‡] $\bigwedge$ and $\bigvee$ are monotonic functions of two arguments, meaning that if $w\le x$ and $y\le u$, then $w\bigwedge y \le x\bigwedge u$ and $w\bigvee y \le x\bigvee u$.

If we can find very precise such x_j's, then our compiler can tailor its code to accomodate only those types of values which the x_j's indicate might arise during an actual program execution.

We know how to make type inferences when passing through the nodes, in the sense that our functions f_j and b_j indicate how the variables of our program take on new types as each statement is executed. Making use of the information contained in the flow graph of P, we now define functions F and B which characterize forward and backward propagation, respectively, of type information throughout the program

F is a function from $[z\rightarrow T]^n$ to $[z\rightarrow T]^n$ given by

$$[F(x_1,x_2,\ldots,x_n)]_j = \bigvee_{m\in pred(j)} f_m(x_m).$$

where $pred(j) = \{ m \mid m\rightarrow j$ is an edge in the flow graph of $P \}$

Intuitively, if $y_j=[F(x_1,\ldots,x_n)]_j$, then y_j expresses what can be deduced about the types of the values of the variables at node j given the types of the variables at the nodes that flow into j, as described by vector $x = (x_1,\ldots,x_n)$.

The following matrix notation gives us an alternative way of writing the expression $F(x)$. Consider F as an $n\times n$ matrix of functions, F_{jm}, each $F_{jm}:[z\rightarrow T]\rightarrow[z\rightarrow T]$, where

$$F_{jm} = \begin{cases} f_m & \text{, if } m\rightarrow j \text{ is an edge of } P \\ \underline{0} & \text{, otherwise} \end{cases}$$

and $\underline{0}(t)=0$, for all $t\in[z\rightarrow T]$ ‡

So $F(x)=F'x$ is the inner product of an $n\times n$ matrix of functions from $[z\rightarrow T]$ to $[z\rightarrow T]$ and an n-vector of mappings in $[z\rightarrow T]$, with function application corresponding to scalar multiplication and the lattice function $\bigvee:[z\rightarrow T]\times[z\rightarrow T]\rightarrow[z\rightarrow T]$ corresponding to scalar addition.

‡ 0 stands both for the least element of the lattice $[[z\rightarrow T]\rightarrow[z\rightarrow T]]$ and for the least element of $[z\rightarrow T]$.

Similarly we define $B:[z\rightarrow T]^n\rightarrow[z\rightarrow T]^n$ by,

$$[B(x_1,x_2,\ldots,x_n)]_m = \bigvee_{j\in succ(m)} b_m(x_j),$$

where $succ(m) = \{ j \mid m\rightarrow j$ is an edge in the flow graph of $P \}$

Intuitively, if $y_m=[B(x_1,\ldots,x_n)]_m$, then y_m expresses what can be deduced about the types of the values of the variables at node m, given the types of the variables at the successors of node m.

Again, we can write $B(x)$ in matrix notation as $B'x$, where

$$B_{mj} = \begin{cases} b_m & \text{, if } m\rightarrow j \text{ is an edge of } P \\ 0 & \text{, otherwise} \end{cases}$$

Notice that both F and B are monotonic functions, since they are each compositions of monotonic functions.

F and B represent two different, but related, type inference systems. The following two lemmas show how either F or B may be used to demonstrate the safeness of a proposed solution.

Lemma 2: Let vectors x and s in $[z\rightarrow T]^n$ be such that $x \geq s\bigwedge F(x)$, and let s be safe. Then x is safe.

Proof: The proof is carried out by a straightforward induction on execution path length. □

A similar lemma holds for B.

Lemma 3: Let x be such that $x\geq s\bigwedge B(x)$, where s is safe, $x,s\in[z\rightarrow T]^n$. Then x is safe.

Proof: The proof is similar to that of Lemma 2. □

Fortunately, the two preceeding lemmas not only give a means of testing the safeness of a solution; but they also suggest a way to compute a safe solution. We shall demonstrate this for the case of the forward type propagation system, F. All of the following may also be carried out in the B system.

Assume we have a safe solution, $s\in[z\rightarrow T]^n$, and we hope to find a better solution x. The fact that any x which satisfies $x \geq s\bigwedge F(x)$ is safe suggests

that we look for the smallest such x, namely that we find the smallest x such that $x=s\wedge F(x)$.

Considering s as fixed, define $F_s:[z\to T]^n\to[z\to T]^n$ by

$$F_s(x) = s\wedge F(x)$$

(Notice that F_s is a monotonic function because $\wedge$ and F are monotonic.) Now let $v=F_s^*(\underline{0})$ be the least fixedpoint of F_s. That is, v is the least element of $[z\to T]^n$ such that $v = F_s(v) = s\wedge F(v)$. By Lemma 2, v is provably safe. Notice that the monotonicity of F_s implies that $F_s^{i+1}(\underline{0})\geq F_s^i(\underline{0})$, for i=0,1,2,.... which also gives us

$$F_s^m(\underline{0})=\bigvee_{i=1,2,\ldots,m} F_s^i(\underline{0})$$

Thus we can write v explicitly as,

$$v = F_s^*(\underline{0}) = \bigvee_{i=1,2,3,\ldots} F_s^i(\underline{0})$$

$F_s^*(0)$ can always be computed in finitely many steps as long as the lattice T has no infinite chains.

Now considering s as the variable, define the function $\sharp:[z\to T]^n\to[z\to T]^n$ by $\sharp(s) = F_s^*(\underline{0})$ or algorithmically by

```
♯(s) :=  {  v ← 0;
         while v ≠ s∧F(v) do
           v ← s∧F(v);
         return v  }
```

We summarize what appears above by:

Lemma 4: If s is safe then ♯(s) is safe.

The following lemma expresses some interesting facts about the function ♯.

Lemma 5:
(a) ♯ is decreasing. i.e., for all x, $\sharp(x)\leq x$.
(b) ♯ is monotonic.
(c) $\sharp^2(x)=\sharp(x)$, i.e., for all x, ♯(x) is a fixedpoint of ♯.
(d) $\sharp(F_x^K(0))=F_x^K(\underline{0})$, for all x, and for all K>0.

Proof:
(a) $\sharp(x)=F_x^*(0)=x\wedge F(\sharp(x)) \leq x$
(b) Let $x\leq y$. We claim $F_x^i(\underline{0})\leq F_y^i(\underline{0})$, for all i . so that $\sharp(x)=F_x^*(\underline{0}) \leq F_y^*(0)=\sharp(y)$. Our proof proceeds by induction on i. For i=0 we have $F_x^i(0)=\underline{0}=F_y^i(0)$. Now assume inductively. $F_x^i(0)\leq F_y^i(0)$. This, along with the monotonicity of F gives us

(b1) $F(F_x^i(\underline{0})) \leq F(F_y^i(\underline{0}))$

Our hypothesis is that $x\leq y$. so by (b1) and the monotonicity of $\wedge$:

(b2)
$F_x^{i+1}(\underline{0})=x\wedge F(F_x^i(\underline{0})) \leq y\wedge F(F_y^i(\underline{0}))=F_y^{i+1}(\underline{0})$

which completes the induction.

(c) Let $y=\sharp(x)=F_x^*(0)$. We claim

(c1) $y \geq F_x^i(\underline{0})$, for all i

and

(c2) $F_y^i(\underline{0}) = F_x^i(\underline{0})$, for all i.

As in part (b), we proceed by induction on i. At i=0 both (c1) and (c2) are trivially true. Now assume the induction hypothesis for (c1). That and the monotonicity of F_x imply that $F_x(y)\geq F_x^{i+1}(\underline{0})$. But since y is a fixedpoint of F_x, we have $y=F_x(y)\geq F_x^{i+1}(\underline{0})$, which completes the induction step for (c1).

To prove (c2), assume the induction hypothesis and apply F_y to get $F_y^{i+1}(\underline{0})=F_y(F_x^i(\underline{0}))=y\wedge F(F_x^i(0))$. But since y is a fixedpoint of F_x we also have $y=F_x(y)=x\wedge F(y)$. So we can write

(c2.1) $F_y^{i+1}(\underline{0}) = x\wedge F(y)\wedge F(F_x^i(\underline{0}))$

Combining the fact that F is monotonic with (c1), we see $F(y)\geq F(F_x^i(\underline{0}))$. By the definition of '$\geq$' this means $F(F_x^i(\underline{0}))=F(y)\wedge F(F_x^i(\underline{0}))$. Thus we can substitute for $F(y)\wedge F(F_x^i(\underline{0}))$ in (c2.1) and get $F_y^{i+1}(\underline{0})=x\wedge F(F_x^i(\underline{0}))=F_x^{i+1}(\underline{0})$, which completes the induction. Finally, notice that (c2) implies that

$$\sharp^2(x)=\sharp(y)=F_y^*(\underline{0})=F_x^*(0)=\sharp(x).$$

To prove (d), let $y=F_x^K(\underline{0})$. Using arguments similar to those given above, we may prove by induction on i that

(d1) $F_y^i(\underline{0}) = F_x^i(\underline{0})$, for $0\leq i<K$

and

(d2) $F_y^i(\underline{0}) = F_x^K(\underline{0})$, for $i\geq K$.

Hence

$$\sharp(y)=F_y^*(\underline{0})=F_x^K(\underline{0}).$$

□

IV A Type Determination Algorithm

A consequence of Lemma 5 is that given any safe solution s, we can apply $\sharp$ to it to get a (possibly) better solution, $\sharp(s)$. But no further applications of $\sharp$ can yield any improvements over $\sharp(s)$.

Of course all of the above arguments work equally well for the backwards inference system - We can construct a $\flat$ function from B by defining $\flat(s) = B_s^*(\underline{0})$, where $B_s(x) = s \wedge B(x)$. And we can state Lemmas 4 and 5' by just substituting the symbol $\flat$ in place of $\sharp$ everywhere that $\sharp$ appears in Lemmas 4 and 5. So given a safe solution s we can compute a (possibly) better solution $\flat(s)$.

Notice that the fact that a given safe solution s cannot be improved by further applications of $\sharp$ or $\flat$ does not imply that it may not be improved by an application of $\flat$ or $\sharp$, respectively. It is easy to demonstrate programs where $\flat\cdot\sharp(s)$ gives a better solution than either $\sharp(s)$ or $\flat(s)$.‡ Given an initial safe solution s, (eg. $\underline{1}$, which is always safe) we can compute

$$\hat{s} = (\flat\cdot\sharp)^*(s) = \flat\cdot\sharp\cdot\ldots\cdot\flat\cdot\sharp(s).$$

In the next section we shall present some general results about monotone functions on lattices that show $\hat{s}$ is, in some sense, an optimal solution to the type determination problem.

V Optimality Results

Given a set of monotone functions, $H=\{h_1,h_2,\ldots,h_n\} \subseteq [L\to L]$, whose members each map a complete lattice L into itself, given L's meet and join functions, $M=\{\bigwedge,\bigvee\} \subseteq [LXL\to L]$, and given a set of initial points, $S=\{s_1,s_2,\ldots,s_m\} \subseteq L$, we would like to study the set of points, C(H,M,S), which can be computed by arbitrarily applying arbitrary compositions of the functions of H and M to the points of S. Just what we mean by "computed" and "arbitrary" will become clear as we proceed in our investigation.

Our motivation is as follows. If each of the points in S represents a safe solution to the type determination problem and each of the functions in H and M preserve safety, then every point in C(H,M,S) will also represent a safe solution. We shall use the theory we develop to prove that $(\flat\cdot\sharp)^*(s)$ is the best solution we can find, given operators $\flat$, $\sharp$, B, F, $\bigwedge$, $\bigvee$, composition and application of functions, and an initial safe solution s

‡ We shall present such a program at the end of this paper.

Since each point in C(H,M,S) is to be computable (in finitely many operations) for each point c in C(H,M,S) there should be some formula e that expresses how to compute c from the sets H, M, and S. So, turning the problem around, we shall first look at a rather large class of formulas, $(\hat{E}\cup\hat{G})$, defined by:

$\hat{E} = \cup\{E^i \mid i=1,2,\ldots\}$

$\hat{G} = \cup\{G^{j,k} \mid j,k=1,2,\ldots\}$

where the E^i's and the $G^{j,k}$'s are given recursively by the rules which follow. Intuitively, the E^i's are vectors of i expressions with values in L, and the $G^{j,k}$'s are functions from L^j to L^k.

(E.i) (variable introduction) If x is a variable name, then x is in E^1.

(E.ii) (concatenation) If $e^i \in E^i$ and $e^{k-i} \in E^{k-i}$, then $(e^i,e^{k-i}) \in E^k$.

(E.iii) (function application) If $g^{j,k} \in G^{j,k}$ and $e^j \in E^j$, then $g^{j,k}(e^j) \in E^k$.

(E.iv) (function closure) If $g^{k,k} \in G^{k,k}$ and $e^k \in E^k$, then both $[g^{k,k}]^*(e^k) \in E^k$ and $[g^{k,k}]^{\bar{*}}(e^k) \in E^k$. (We use * for the greatest lower bound (glb) and $^{\bar{*}}$ for the least upper bound (lub) of an iterated application of $g^{k,k}$ to e^k.)

(G.i) (abstraction) If $e^k \in E^k$ and $y_1,y_2,\ldots,y_j$ are variable names, then $\lambda(y_1,y_2,\ldots,y_j).e^k \in G^{j,k}$.

(G.ii) (function introduction) If $g^{j,k}$ is the name of a function from $L^j \to L^k$, then $g^{j,k} \in G^{j,k}$.

Our intent is that each formula in E^k may be interpreted as an element of L^k, provided we interpret each free variable in a formula as an element of L. Similarly the formulas in $G^{j,k}$ may each be interpreted as functions mapping L^j to L^k. Since we are only given the functions in H and M and the initial lattice points in S to start with, let us consider how to interpret the set of formulas $(\hat{E}\cup\hat{G})[H,M,S]$. The notation $Q[H,M,S]$ stands for the set of expressions e such that

(1) e is a formula in the set Q,
(2) if x is a free variable name in e, then $x \in \{'s_1', 's_2', \ldots, 's_m'\}$, i.e., x names an element of S, and
(3) if g is a function name occuring in e then either $g \in G^{1,1}$ and g names an element of H, or $g \in G^{2,1}$ and g names an element of M.

To interpret a formula, $e \in (E \cup G)[H,M,S]$, we first define a function I, which maps variable names $s_1, \ldots, s_m$ into the corresponding lattice points of S, and which maps function names $h_1, \ldots, h_n$ and $\wedge, \vee$ to the corresponding functions of H and M. Next, we extend I to $\hat{I}$, a function whose domain is $(E \cup G)[H,M,S]$, by recursively defining:

(IE.i) $\hat{I}(x) = I(x)$, if x is a variable name.

(IE.ii) $\hat{I}((e^i, e^{k-i})) = (\hat{I}(e^i), \hat{I}(e^{k-i})) \in L^k$

(IE.iii) $\hat{I}(g^{j,k}(e^j)) = [\hat{I}(g^{j,k})](\hat{I}(e^j)) \in L^k$

(IE.iv.a) $\hat{I}([g^{k,k}]^*(e^k)) =$

$$glb\{[\hat{I}(g^{k,k})]^i(\hat{I}(e^k)) \mid i=0,1,2,\ldots\}$$

(IE.iv.b) $\hat{I}([g^{k,k}]^{\#}(e^k)) =$

$$lub\{[\hat{I}(g^{k,k})]^i(\hat{I}(e^k)) \mid i=0,1,2,\ldots\}$$

(IG.i) $\hat{I}(\lambda(y_1, y_2, \ldots, y_j).e^k)$ is the function $f \in [L^j \to L^k]$, given by:

$f(t_1, t_2, \ldots, t_j) = \hat{J}(e^k)$, where $\hat{J}$ is the extension of an interpretation J that is given by:

$$J(e) = \begin{cases} t_i, & \text{if } e = \text{the variable } y_i, \text{ for some } i \\ I(e), & \text{if } e = \text{some other variable or function name} \end{cases}$$

Note that this definition is not circular, since e^k has fewer instances of abstraction than $\lambda(y_1, \ldots, y_j).e^k$.

(IG.ii) $\hat{I}(g) = I(g)$, if g is a function name.

The correspondence between these rules for interpreting formulas, and the rules given above for building formulas should be obvious - we are just saying that a formula is interpreted by interpreting each of the parts of which it was built. In the case of rules (IE.iv.a) and (IE.iv.b), it should be noted that the existence of greatest lower and least upper bounds is guaranteed by our assumption that L is a complete lattice (and hence so is each L^k). Thus $\hat{I}$ maps each formula in $(E \cup G)[H,M,S]$ into a member of the set

$$(\cup\{L^i \mid i=1,2,\ldots\}) \cup (\cup\{[L^j \to L^k] \mid j,k=1,2,\ldots\}).$$

We can now define $C(H,M,S) = \{\hat{I}(e) \mid e \in (E \cup G)[H,M,S]\}$. Observe that because of rules (E.ii), (E.iii) and (E.iv), C(H,M,S) is necessarily closed under finite Cartesian products, function application, and function closures. It is also true that C(H,M,S) contains the projection functions, $p_i^k: L^k \to L$, since rule (G.i) allows us to write formulas of the form $\lambda(x_1, \ldots, x_k).x_i$. Also notice that if $g: L^i \to L^j \in C(H,M,S)$ and $f: L^j \to L^k \in C(H,M,S)$ then there exist formulas $e_g \in G^{i,j}$ and $e_f \in G^{j,k}$ such that $g = \hat{I}(e_g)$, $f = \hat{I}(e_f)$, and $\lambda(x_1, \ldots, x_i).e_f(e_g(x_1, \ldots, x_i))$ is a formula in $G^{i,k}$ whose interpretation is just f·g. Thus C(H,M,S) is closed under function composition.

The reader should now be convinced that C(H,M,S) is the set of all points and functions which can be computed from H, M, and S by arbitrary function composition and application.

Two points should be made here. First, we have introduced the concatenation construction (E.ii) (which leads to closure under finite Cartesian products) to capture the notion that during a computation we may separately compute and store several different values which may be recombined later by a further computation. Second, we have introduced the two forms of function closure (E.iv) to capture the notion that we may apply a particular function arbitrarily many times, in a iterative fashion, accumulating intermediate results in a meet or join, halting only when that meet or join reaches a minimum or maximum value, respectively. Notice that if the underlying lattice L satisfies the finite chain condition, then function closure can be effectively computed.

We would now like to further investigate the properties of C(H,M,S). Our first result says (roughly speaking) that all functions in C(H,M,S) are monotonic and that everything in C(H,M,S) monotonically depends on the values in the sets H, M, and S.

Recall that the domain of an interpretation I for the single symbol formulas in $(\dot{E}UG)[H,M,S]$ is the set of names for the elements of H, M, and S. We shall say that I' is an alternative interpretation for the single symbol formulas in $(\dot{E}UG)[H,M,S]$ if I' has the same domain as I, but I' maps the names of elements of S into (possibly) different values in L, maps the names of elements of H into (possibly) different monotonic functions in $[L\to L]$, and maps the names of elements of M into (possibly) different monotonic functions in $[L\times L\to L]$. A partial ordering on alternative interpretations is defined by saying $I'\leq I$ if and only if for all e in the domain of I, $I'(e)\leq I(e)$.

Now we can state:

Theorem 1: Let I' and I be alternative interpretations for the single symbol formulas in $(\dot{E}UG)[H,M,S]$, such that $I'\leq I$. Then

(1) for all $e\in(\dot{E}UG)[H,M,S]$, $\hat{I}'(e)\leq\hat{I}(e)$.
(2) for all $g\in G[H,M,S]$, $\hat{I}(g)$ is a monotonic function.

Proof: The proof is a straightforward induction on the structural complexity (number of applications of rules (E.i)-(E.iv), (G.i) and (G.ii) used in the construction) of the formula e. □

For a given interpretation I, we say that formulas e_1 and e_2 are equivalent if $\hat{I}(e_1)=\hat{I}(e_2)$. It is easy to see that $(\dot{E}UG)[H,M,S]$ is partitioned into equivalence classes by this relation and that the set of these equivalence classes is isomorphic to C(H,M,S). Since each member of C(H,M,S) has (at least) one formula in $(\dot{E}UG)[H,M,S]$ that represents it, in the text that follows we shall find it convenient to blur the distinction between members of C(H,M,S) and their representative formulas. All formulas will be understood to be interpreted by $\hat{I}$, unless we indicate otherwise.

Notice that there is a natural partial ordering on the elements of C(H,M,S); for $c_1,c_2\in C(H,M,S)$, $c_1\leq c_2$ if and only if either $(c_1,c_2\in L^k$ for some k and $c_1<c_2)$ or $(c_1,c_2\in[L^k\to L^j]$ and $c_1\leq c_2)$. Our next result shows that minimal and maximal elements of C(H,M,S) exist and can be represented by simple formulas.

First, for notational convenience, we define:

$$\bigwedge^k = \text{k-wise meet} = \lambda(y_1,y_2,\ldots,y_k).y_1\wedge y_2\wedge\ldots\wedge y_k$$

$$\bigvee^k = \text{k-wise join} = \lambda(y_1,y_2,\ldots,y_k).y_1\vee y_2\vee\ldots\vee y_k$$

$$D^k = \text{k-duplicator} = \lambda y.(y,y,\ldots,y)\in[L\to L^k]$$

$$h = \text{meet of } H = \lambda y.h_1(y)\wedge h_2(y)\wedge\ldots\wedge h_n(y)\wedge y$$

$$\bar{h} = \text{join of } H = \lambda y.h_1(y)\vee h_2(y)\vee\ldots\vee h_n(y)\vee y$$

$s = \text{vector of } S = (s_1,s_2,\ldots,s_m)$

$$\underline{\min}[H,M,S] = (h)^*(\textstyle\bigwedge^m(s))$$

$$\underline{\min}^k[H,M,S] = D^k(\underline{\min}[H,M,S])$$

$$\underline{\max}[H,M,S] = (\bar{h})^*(\textstyle\bigvee^m(s))$$

$$\underline{\max}^k[H,M,S] = D^k(\underline{\max}[H,M,S])$$

$$\underline{\min}^{j,k}[H,M,S] = \lambda(x).D^k((h)^*(\textstyle\bigwedge^{m+j}(s,x))), \quad \text{where } x=(x_1,\ldots,x_j)$$

$$\underline{\max}^{j,k}[H,M,S] = \lambda(x).D^k((\bar{h})^*(\textstyle\bigvee^{m+j}(s,x))), \quad \text{where } x=(x_1,\ldots,x_j)$$

We shall drop the '[H,M,S]' suffix, except where it is important to emphasize that the various min's and max's are defined in terms of the sets H, M and S.

To prove the next theorem we will need:

Lemma 6: For all j,k, $\underline{\max}^{j,k}(\underline{\max}^j[H,M,S]) = \underline{\max}^k[H,M,S]$ and $\underline{\min}^{j,k}(\underline{\min}^j) = \underline{\min}^k$.

Proof: First notice that $\underline{\max}\geq s_i$ for each $s_i\in S$, because

$$\underline{\max}=(\bar{h})^*(\textstyle\bigvee^m(s))\geq\bigvee^m(s)\geq s_i.$$

Also notice that h is monotone since it is in C(H,M,S), and that by construction $\bar{h}$ is increasing, i.e., $\bar{h}(x)\geq x$, for all x. So

$$\underline{\max}^{j,k}(\underline{\max}^j) = D^k((\bar{h})^*(\textstyle\bigvee^{m+j}(s,\underline{\max}^j))) = D^k(\bar{h})^*(\underline{\max})) = D^k((\bar{h})^*((\bar{h})^*(\textstyle\bigvee^m(s)))) \ddagger =$$

‡ Note that, for all x, since h is increasing, the elements of $\{\bar{h}^i(x)\mid i=0,1,2,\ldots\}$ form a chain. And since L is assumed to satisfy the finite chain condition, all monotonic functions are continuous [5]. Therefore for all x,

$$(\bar{h})^*((\bar{h})^*(x)) = (\bar{h})^*(x).$$

$D^K((\bar{h})^*(\vee^m(s))) = \underline{max}^K$. A similar argument shows the result for $\underline{min}^K$. ∎

Now we can state:

Theorem 2:

(a) Let $c \in C(H,M,S) \cap L^K$, for some k. Then $\underline{min}^K[H,M,S] \leq c \leq \underline{max}^K[H,M,S]$.

(b) Let $c \in C(H,M.S) \cap [L^j \to L^K]$, for some j,k. Then $\underline{min}^{f^{j,K}}[H,M,S] \leq c \leq \underline{max}^{f^{j,K}}[H,M,S]$.

Proof: Since each $c \in C(H,M,S)$ has a representative formula we can carry out a proof by induction on the structure of these formulas. As in the case of the previous theorem, there will be an argument for each of the rules (E.i)-(E.iv) and (G.i)-(G.ii). For the sake of brevity we only present the argument for rule (E.iii). After seeing this, the manner in which the rest of the proof could be carried out will become obvious.

Assume $c \in C(H,M,S) \cap L^K$ can be written as $g(e)$, where $g \in C(H,M,S) \cap [L^j \to L^K]$ and $e \in C(H,M,S) \cap L^j$ and both g and e have smaller minimal formulas than does c. By the induction hypothesis $e \leq \underline{max}^j[H,M,S]$. Hence by the monotonicity of g, $g(e) \leq g(\underline{max}^j)$. The induction hypothesis also guarantees that $g \leq \underline{max}^{f^{j,K}}[H,M,S]$. So $g(\underline{max}^j) \leq \underline{max}^{f^{j,K}}(\underline{max}^j)$, which equals $\underline{max}^K[H,M,S]$ by the preceeding lemma. A similar argument shows that $\underline{min}^K[H,M,S] \leq c$. ∎

Now, let us apply Theorem 2 to the solution of Section IV. Let $d=\{\Phi,\Psi,F,B\}$, $M=\{\wedge,\vee\}$, and $S = \{s\}$ for some safe solution $s \in [z \to \mathbb{T}]^n$, where $z=\{Z_1,Z_2,\ldots,Z_n\}$ is the set of variables of an n-node flow graph. We can now show that

Theorem 3: The best safe solution in $C(H,M,S)$ over the lattice $[z \to \mathbb{T}]^n$ is $\hat{s} = (\Psi \cdot \Phi)^*(s) = (\Phi \cdot \Psi)^*(s)$.

Proof: By the previous theorem with $L=[z \to \mathbb{T}]^n$, we know that the smallest element in $C(H,M,S) \cap [z \to \mathbb{T}]^n$ is $\underline{n}^*(s)$. But notice that because Ψ is decreasing, and B_x is monotonic, for any x we have $\Psi(x) = B_x^*(\underline{0}) = B_x(\Psi(x)) \leq B_x(x) = x \wedge B(x)$. Similarly, $\Phi(x) \leq x \wedge F(x)$. Moreover, because Φ is decreasing, $\Phi(\Psi(x)) \leq \Psi(x)$, and because Ψ is decreasing, and Φ is monotonic, $\Phi(\Psi(x)) \leq \Phi(x)$. Combining these facts with a little lattice algebra, we get $\Phi(\Psi(x)) \leq \Phi(x) \wedge \Psi(x) \wedge F(x) \wedge B(x) \wedge x$, which equals $\underline{n}(x)$, for all x. Therefore $\Phi \cdot \Psi \leq \underline{n}$, which implies that $\hat{s} = (\Phi \cdot \Psi)^*(s) \leq \underline{n}^*(s)$, which equals $\underline{min}[d,M,S]$. But $(\Phi \cdot \Psi)^*(s) \in C(H,M,S) \cap [z \to \mathbb{T}]^n$. Hence $\hat{s} = \underline{min}[H,M,S]$. Similarly $(\Psi \cdot \Phi)^*(s) = \underline{min}$. ∎

Since $(\Psi \cdot \Phi)$ is a decreasing function, no confusion will result from writing $(\Psi \cdot \Phi)^*(s)$ as $(\Psi \cdot \Phi)^*(s)$. A similar remark applies for $\Phi \cdot \Psi$.

Optimality of Φ and Ψ

Observe that Φ is not directly expressible in terms of F and B, in the sense that, unless $\underline{0}$ is in S, the formula for Φ, which is $\lambda s.((\lambda y.s \wedge F(y))^*(\underline{0}))$, is not in $C(\{F,B\},\{\wedge,\vee\},S)$. A similar statement applies to Ψ. Because $\underline{0}$ is generally not a safe solution, we do not wish to introduce $\underline{0}$ into S. To explore the properties of formulas like those for Φ and Ψ we shall define a new class of computable objects $A(H,M,S)$. $A(H,M,S)$ is much like $C(H,M,S)$, except that in defining the set of underlying formulas and their interpretations, we add rule (E.v) and its interpreting rule (IE.v):

(E.v) (least fixedpoint formation) If
(1) $g^{j,K} \in G^{j,K}$,
(2) $g^{j,K}$ has no free occurrences of variables $y_1,\ldots,y_K$,
(3) $e_1,\ldots,e_j \in E^1$, and
(4) n names a function in H,
then let $f = \lambda(y_1,\ldots,y_K).g^{j,K}(n(e_1),\ldots,n(e_j))$.
(Note that f is a member of $G^{K,K}$).
Then $f^*(\underline{0}^K) \in E^K$, where $\underline{0}^K = (\underline{0},\ldots,\underline{0})$ represents the least element of L^K.

(IE.v) $I(f^*(\underline{0}^K)) = \text{lub}\{[I(f)]^i(\underline{0}^K) \mid i=0,1,2,\ldots\}$, which is the least fixedpoint of the function $I(f)$.

We are very careful in specifying the form of function f in rule (E.v) because we do not want unsafe solutions to enter into $A(H,M,S)$. In particular, the application of a function in H to each argument of $g^{j,K}$ is one way to assure safety in the presence of a least fixedpoint operator, although it is not the only conceivable way.

Formally, we let R be the set of all formulas that can be built by recursively applying rules (E.i)-(E.v), (G.i) and (G.ii); let I be the natural interpretation for the names of the elements of H, M, and S; and let $\hat{I}$ be the extension of I given by rules (IE.i)-(IE.v), (IG.i) and (IG.ii). Then we define $A(H,M,S) = \{ I(e) \mid e \in R[H,M,S] \}$. As we did while discussing the elements of C(H,M,S), we shall usually denote the members of A(H,M,S) by their representative formulas, understood to be interpreted by I.

In particular, note that

$$f_s = \lambda y.s \wedge F(y) \in A(\{F\},M,\{s\})$$

and that $\Phi = \lambda s.F_s^*(\underline{0})$ can be written as

$$\Phi = \lambda s.([\lambda y.[\lambda x.s \wedge x](F(y))]^*(\underline{0})),$$

which is in a form admitted by rules (E.i)-(E.v), (G.i) and (G.ii). So $\Phi \in A(\{F\},M,S)$, for any S, since there are no free variables in the formula for Φ. Similar formulas can be given to demonstrate that B_s and Φ are in $A(\{B\},M,\{s\})$.

The following lemma expresses the observation that all functions in A(H,M,S) are monotonic and that all objects in A(H,M,S) depend monotonically on I.

Lemma 7: Let I' and I be alternative interpretations for the single symbol formulas in R[H,M,S], such that $I' \leq I$. Then
(1) for all $e \in R[H,M,S]$, $I'(e) \leq \hat{I}(e)$.
(2) for all $g \in R[H,M,S]$ such that g is in a form given by either rule (G.i) or (G.ii), I(g) is a monotonic function.

Proof: The proof is essentially the same as that of Theorem 1. ∎

An immediate consequence of Lemma 7 is that Φ and Φ are monotonic functions; thus we have an alternative way of demonstrating a fact that we stated earlier as part (b) of Lemma 5

We shall now show that, in the case where $H=\{F\}$, $M=\{\wedge,\vee\}$, $S=\{s_1,\ldots,s_m\} \subseteq$ lattice L, and $F \in [L \to L]$, then A(H,M,S) has minimal elements. As usual, all that follows will still hold if B and Φ are uniformly substituted for F and Φ, respectively. Just as we defined min and minf we now define:

$$\underline{low}[F,M,S] = \Phi(\wedge^m(s))^{\ddagger}$$

$$\underline{low}^K[F,M,S] = D^K(\underline{low}[F,M,S])$$

$$\underline{low}^{f^{j,K}}[F,M,S] =$$

$$\lambda(x).D^K(\Phi(\wedge^{m+j}(s,x))), \quad \text{where } x=(x_1,\ldots,x_j)$$

Corresponding to Lemma 6, we have:

Lemma 8: For all j,k, $\underline{low}^{f^{j,K}}(\underline{low}^j[F,M,S]) = \underline{low}^K[F,M,S]$.

Proof: The proof is much like that of Lemma 6, the key points being that Φ is decreasing and monotonic and that $\Phi' \Phi = \Phi$. But these facts are given by parts (a), (b), and (c) of Lemma 5. ∎

Theorem 4:
(a) Let $a \in A(F,M,S) \cap L^K$, for some k. Then $a \geq \underline{low}^K[F,M,S]$.

(b) Let $a \in A(F,M,S) \cap [L^j \to L^K]$. Then $a \geq \underline{low}^{f^{j,K}}[F,M,S]$.

Proof: The proof closely follows that of Theorem 2, except that we must present a new argument corresponding to the new rule (E.v).

Assume $a \in A(F,M,S) \cap L^K$ has a representative formula of the form $f^*(\underline{0}^K)$, where $f = (\lambda(y_1,\ldots,y_K).g^{j,K}(F(e_1),\ldots,F(e_j)))$ $\in A(F,M,S) \cap [L^K \to L^K]$, and $g^{j,K}$, $e_1,\ldots,e_j$ are each of a form admitted by rule (E.v). By the induction hypothesis, $g^{j,K} \geq \underline{low}^{f^{j,K}}[F,M,S]$ and $e_i \geq \underline{low}[F,M,S']$, for $1 \leq i \leq j$, where $S' = S \cup \{y_1,\ldots,y_K\}$. Therefore, because all functions belonging to A(F,M,S) are monotonic,

$$(1)\quad f \geq \lambda(y).[\underline{low}^{f^{j,K}}[F,M,S]](F(\underline{low}[F,M,S']),\ldots,F(\underline{low}[F,M,S'])),$$

where $y = (y_1,\ldots,y_K)$.

The right hand side of (1) can easily be shown to be equal to $\lambda(y).D^K \cdot \Phi \cdot F_s \cdot \Phi(\wedge^{1+K}(\underline{s},y))$, where $s = \wedge^m(s)$.

We claim $f^i(\underline{0}^K) \geq D^K \cdot F_s^i(\underline{0})$, for all $i \geq 0$. This is shown by induction on i. For i=0 the claim is trivially true. Now by the induction hypothesis and the monotonicity of f, $f^{i+1}(\underline{0}^K) > f(D^K \cdot F_s^i(\underline{0}))$. This and inequality (1) yield

$$(2)\quad f^{i+1}(\underline{0}^K) \quad D^K \cdot \Phi \cdot F_s \cdot \Phi(\wedge^{1+K}(s, D^K \cdot F_s^i(\underline{0}))).$$

‡ We shall use F in place of {F}, when it is convenient and the context makes our meaning clear. Also, s here stands for a vector of the m elements of S.

The right hand side of (2) may be reduced to $D^K \cdot \Phi \cdot F_s \cdot \Phi \cdot F_s^i(\underline{0})$, which is seen to equal $D^K \cdot F_s^{i+1}(\underline{0})$, by twice invoking part (d) of Lemma 5. This completes the induction step.

So we have $a = \bar{F}^*(\underline{0}^K) = \text{lub}\{F^i(\underline{0}^K)\} \geq \text{lub}\{D^K \cdot F_s^i(\underline{0})\} = D^K(\text{lub}\{F_s^i(\underline{0})\}) = D^K \cdot \Phi(s) = \text{low}^K[F,M,S]$. ∎

Theorem 4 says that given some safe solution $s \in [z \to T]^n$, the best safe solution in $A(\{F\},\{\wedge,\vee\},\{s\})$ is $\Phi(s)$. Or, in other words, given type inference function F, computing $\Phi(s)$ is an optimal way to use F to improve a safe solution s. We remind the reader that similar remarks can be made about Φ.

VI Comparison with other techniques

Other researchers have proposed alternative methods for computing good safe solutions to the type finding problem. In this section we express some of these methods in the notation which we have developed above. We then prove the inequalties which indicate that our method yields better solutions.

Jones and Muchnick [J] construct systems of equations which correspond to forward and backward inference of types. In our notation their backward system corresponds very closely to‡

$$y = B(y)$$

and their forward system is just

$$x = F_y(x).$$

They suggest solving the backward system for its maximal fixpoint, substituting this into the forward system and solving for the minimal fixpoint. That is, they set

$$y_0 = B^*(\underline{1})$$

and

$$x_0 = F^*_{y_0}(0) = \Phi(y_0)$$

Our technique is somewhat more general in that we can easily incorporate any additional information provided by a given safe solution s which might, for example, be derived from user declarations within the program. Also notice that $\Phi \cdot \Phi(\underline{1}) = \Phi(B^*(\underline{0})) \leq \Phi(B^*(\underline{1}))$, which is Jones and Muchnick's solution. So our technique is at least as powerful as that of [J].

Tenenbaum's [T] idea is to compute a safe solution in two stages. First an initial solution x_0 is computed by considering forward type inferences. Then the initial solution is improved by considering both forward and backward type inferences simultaneously. We can express this in our notation as follows. Let $x_0 = \Phi(\underline{1}) = F^*(\underline{0})$ be the initial safe solution, and define function $G:[z \to T]^n \to [z \to T]^n$ by $G(x) = F(x) \wedge B(x) \wedge x$. It is easy to see that x safe implies that $x \wedge F(x) \wedge B(x) = G(x)$ is safe. It is also easy to see that $G(x) \leq x$ for all x, so that applying G repeatedly to a safe solution can only improve it. Thus, Tenenbaum's final solution is $y_0 = G^*(x_0)$. But in the previous section we showed that $\Phi \cdot \Phi(x) \leq \Phi(x) \wedge \Phi(x) \wedge F(x) \wedge B(x) \wedge x$, which is $\leq G(x)$. Therefore $(\Phi \cdot \Phi)^*(\Phi(\underline{1})) \leq G^*(\Phi(\underline{1})) = y_0$.

Let us also observe that the operators Φ and Φ are not necessary to produce a solution that is superior to what Tenenbaum proposes. Using only the functions F and B, we can obtain the safe solution $G^*(F^*(\underline{0}) \wedge B^*(\underline{0}))$, which is never worse than Tenenbaum's solution and is, in fact, minimal in $C(H,M,S)$ when $H=\{F,B\}$, $M=\{\wedge,\vee\}$, and $S=\{F^*(\underline{0}),B^*(\underline{0})\}$.

Thus our technique may produce stronger assertions about variable types than either Jones and Muchnick's or Tenenbaum's approach.

VII An example

In this section we present a program fragment and, using the definitions and terminology developed above, we compute $(\Phi \cdot \Phi)^*(\underline{1})$. Although our example is extremely simple, it is interesting in that it demonstrates that the inequalities presented in the previous section may hold strictly.

Our programming language is designed to manipulate character strings and scalar numbers. We have four operators:

- in (input) takes no arguments and always returns the next item from the input file.
- 5 takes no arguments and returns the integer five.
- fl(y) returns the greatest integer which is $\leq$ y if y is a number, and returns a string which is the

---

‡ In their paper, Jones and Muchnick also suggest the possibility of computing B as $[B(\underline{x})]_m = \bigwedge_{j \in succ(m)} b_m(x_j)$, rather than using the join we proposed in Section III. We do not consider this version of their algorithm; as pointed out in [J], it can lead to incorrect determination of types except under very strict assumptions about program behavior.

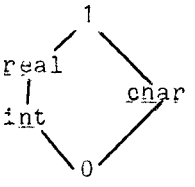
translation of y to lower case if y is a character string.

$\oplus(x,y)$ returns the sum of x and y if both x and y are numbers, returns the concatenation of x and y if both x and y are character strings, and is otherwise undefined.

To form our lattice of types, we choose the basic types:

real = the set of real numbers
int = the set of integers
char = the set of all character strings

and extend to a lattice, T, which is shown diagramatically by:



The type functions, $\{T^j_{op}\}$ are defined by:

$T^0_{in} = 1$

$T^0_5 = int$

y	$T^0_{fl}(y)$
1	1
real	int
int	int
char	char
0	0

$T^1_{fl}(x,y)$

$y \backslash x$	1	real	int	char	0
1	1	real	real	char	0
real	real	real	real	0	0
int	int	int	int	0	0
char	char	0	0	char	0
0	0	0	0	0	0

$T^0_{\oplus}(y_1,y_2)$

$y_1 \backslash y_2$	1	real	int	char	0
1	1	real	real	char	0
real	real	real	real	0	0
int	real	real	int	0	0
char	char	0	0	char	0
0	0	0	0	0	0

$T^1_{\oplus}(1,y_1,y_2)$

$y_1 \backslash y_2$	1	real	int	char	0
1	1	real	real	char	0
real	real	real	real	0	0
int	int	int	int	0	0
char	char	0	0	char	0
0	0	0	0	0	0

$T^1_{\oplus}(real,y_1,y_2)$

$y_1 \backslash y_2$	1	real	int	char	0
1	real	real	real	0	0
real	real	real	real	0	0
int	int	int	int	0	0
char	0	0	0	0	0
0	0	0	0	0	0

$T^1_{\oplus}(int,y_1,y_2)$

$y_1 \backslash y_2$	1	real	int	char	0
1	real	real	int	0	0
real	real	real	int	0	0
int	int	int	int	0	0
char	0	0	0	0	0
0	0	0	0	0	0

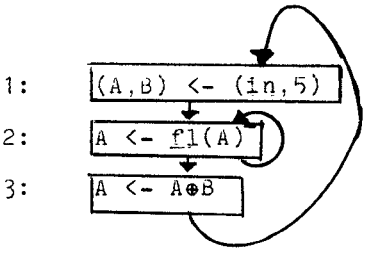
$T^1_{\oplus}(char,y_1,y_2)$

$y_1 \backslash y_2$	1	real	int	char	0
1	char	0	0	char	0
real	0	0	0	0	0
int	0	0	0	0	0
char	char	0	0	char	0
0	0	0	0	0	0

Also, $T^1_{\oplus}(0\ y_1,y_2) = 0$, for all y_1, y_2

and $T^2_{\oplus}(x,y_1,y_2) = T^1_{\oplus}(x,y_2,y_1)$, for all x.

We shall analyze the following program, where 1 is the SP node.



Forward inference functions $f_1,f_2,f_3:T^2 \to T^2$ for statements 1, 2, and 3 are given by:

$[f_1(t_A,t_B)]_A = T^0_{in}$

$[f_1(t_A,t_B)]_B = T^0_5$

$$[f_2(t_A,t_B)]_A = T^0_{f1}(t_A)$$

$$[f_2(t_A,t_B)]_B = t_B$$

$$[f_3(t_A,t_B)]_A = T^0_{\oplus}(t_A,t_B)$$

$$[f_3(t_A,t_B)]_B = T^2_{\oplus}(1,t_A,t_B)$$

The subscripts A and B reference the components of a vector $\in T^2$ which describe variables A and B, respectively.

The matrix of Φ, the forward propagation function is:

$$\Phi = \begin{matrix} \underline{0} & \underline{0} & f_3 \\ f_1 & f_2 & \underline{0} \\ \underline{0} & f_2 & \underline{0} \end{matrix}$$

Backward inference functions $b_1, b_2, b_3 : T^2 \rightarrow T^2$ are given by:

$$[b_1(t_A,t_B)]_A = 1$$

$$[b_1(t_A,t_B)]_B = 1$$

$$[b_2(t_A,t_B)]_A = T^1_{f1}(t_A,1)$$

$$[b_2(t_A,t_B)]_B = t_B$$

$$[b_3(t_A,t_B)]_A = T^1_{\oplus}(t_A,1,t_B)$$

$$[b_3(t_A,t_B)]_B = T^2_{\oplus}(t_A,1.t_B)$$

The matrix of the backward propagation function is:

$$B = \begin{matrix} \underline{0} & b_1 & \underline{0} \\ \underline{0} & b_2 & b_2 \\ b_3 & \underline{0} & \underline{0} \end{matrix}$$

We can now compute:

	$\Phi(\underline{1})$		$G^*(\Phi(\underline{1}))$	
	A	B	A	B
1:	real	int	real	int
2:	1	int	1	int
3:	1	int	real	int

	$\Psi \Phi(\underline{1})$		$\Phi \Psi \Phi(\underline{1})$	
	A	B	A	B
1:	real	int	int	int
2:	real	int	real	int
3:	real	int	int	int

We can also compute that:

$\Psi(\underline{1}) = 1$

$(\Psi \cdot \Phi)^*(\underline{1}) = \Phi \cdot \Psi \cdot \Phi(\underline{1})$

So we have the following relations:

$(\Psi \cdot \Phi)^*(\underline{1}) < \Psi \Phi(\underline{1}) < G^*(\Phi(\underline{1})) < \Phi(\underline{1}) = \Phi \cdot \Psi(\underline{1}) = \Phi \cdot B^*(\underline{1})$

Thus our proposed solution is strictly better than either Tenenbaum's solution or Jones and Muchnick's solution‡ on this example.

References

[A] Aho, A. V., and J. D. Ullman, The Theory of Parsing, Translation, and Compiling: Vol. II, Compiling, Prentice-Hall, Englewood Cliffs, N. J., 1973

[B] Bauer, A. M. and H. J. Saal, 'Does APL really need run-time checking?', Software - Practice and Experience, Vol. 4, 1974, pp. 129-138

[D] Donnellan, T., Lattice Theory, Pergamon Press

[G] Graham, S. L. and M. Wegman, 'A fast and usually linear algorithm for global flow analysis', JACM, Vol. 23, No. 1, January 1976, pp. 172-202

[J] Jones, N. and S. Muchnick, 'Binding time optimization in programming languages', Third ACM Symposium on Principles of Programming Languages, 1976, pp. 77-94

[H] Hecht, M. S. and J. D. Ullman, 'Analysis of a simple algorithm for global flow problems', Proceedings of ACM Symposium on Principles of Programming Languages, 1973, pp. 207-217

[Ka] Kam, J. B. and J. D Ullman, 'Monotone Data Flow Analysis Frameworks', Acta Informatica, Vol. 7, January 1977, pp. 305-317

[Ki] Kildall, G. A., 'A unified approach to global program optimization', Proceedings of ACM Symposium on Principles of Programming Languages, 1973, pp. 194-206

[M] Muchnick, S. S., Private communication, August 1977

[Sc] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N. J., 1973

‡ Muchnick [M] points out that for TEMPO, the language for which the [J] algorithm was developed, constraints on the semantics of operators guarantee that $(\Psi \cdot \Phi)^*(1)$ is the same as $\Phi \cdot B^*(\underline{1})$.

[S] Scott, D., Data Types as Lattices, Unpublished lecture notes, Mathematical Centre, Amsterdam, June 1972, see also a paper of the same name in SIAM Journal of Computing, Vol. 5, No. 3, September 1976, pp. 522-587

[T] Tenenbaum, A., Type Determination for Very High Level Languages, Report NSO-3, Courant Institute of Mathematical Sciences, New York University, 1974