1. Equations for type determination by Tenenbaum's 'backward' method.

In his thesis (hereinafter cited as TT) A. Tenenbaum develops two methods of typefinding, a 'forward' and a 'backward' method, which supplement each other. The 'forward' method is based on conventional data-flow analysis. The 'backward' technique uses a rather ad hoc approach based upon a notion of 'program tree'. The efficiency of this latter method, especially when applied to large programs, seems questionable; for this reason, the present note will suggest an alternate technique which can be used in connection with 'backward' typefinding. The technique to be suggested lies closer to conventional data-flow analysis than does the 'tree' approach of TT. Moreover, our new technique seems easier to develop in a 'cross subroutine' version.

In what follows, we use the terminology introduced in TT, except that we refer to 'ovariables' and 'ivariables' instead of (variable) 'defs' ('definitions') and 'uses'.

Let $P$ be a program, schematized into basic blocks in the usual way. We introduce a number of mappings. Let $oi$ be an ivariable or ovariable occurrence of the variable $v$. Then $bfrom(oi)$ is the set of all ivariable and ovariable occurrences of $v$ from which $oi$ can be reached along a path clear of occurrences of $v$. The set $bfromexit$ is the union over $v$ of the set of all ovariable and ivariable occurrences of $v$ from which a program exit or redefinition of $v$ may be reached along a path clear of occurrences of $v$. The set $ffrom(oi)$ is the set of all ivariable occurrences of $v$ which can be reached from $oi$ along a path free of occurrences of $oi$. Note that the respective functions $bfrom$ and $ffrom$ rather resemble the use-to-definition map $ud$ and the definition-to-use map $du$ of conventional data-flow analysis; they can be calculated by a similar method, to be described in more detail below.
Suppose now that the functions \( f_{from} \) and \( b_{from} \) have been calculated. Then in a typefinding algorithm (like that of TT, pp. 88-89) which uses both 'forward' and 'backward' information, the following relationships can be exploited:

A: if \( o_i \) is an ivariable occurrence of a variable \( v \), then the type \( \text{typ}(o_i) \) associated with \( o_i \) is the conjunction of:

- \( \text{Ai: the types associated with all ovariables which can supply the value of } o_i \); and:

- \( \text{Aii: the type } \text{backtype}(o_i) \) determined by the manner in which \( o_i \) is used. This type is a function both of the operation \( \text{op} \) applied to \( o_i \) and the type information available for the output variable of \( \text{op} \), and:

- \( \text{Aiii: if } o_i \text{ belongs to } b_{fromexit}, \text{ then nil; else the disjunction of the types associated with all the elements of } f_{from}(o_i). \)

B: if \( o_i \) is an ovariable occurrence of \( v \), then the type \( \text{typ}(o_i) \) associated with \( o_i \) is that determined by the types associated with the input arguments of \( o_i \).

These relationships are summarized in the following equations:

\[
\begin{align*}
\text{(2a)} \quad & \text{for ovariables: } \text{typ}(o) = \text{forward}(o); \\
\text{(2b)} \quad & \text{for ivariables:} \\
& \quad \text{if } i \in b_{fromexit} \text{ then} \\
& \quad \quad [\text{dis: } o \in \text{ud}(i)] \text{typ}(o) \text{ con } \text{backtype}(i) \\
& \quad \quad \text{else}[\text{dis: } o \in \text{ud}(i)] \text{typ}(o) \text{ con } \text{backtyp}(i) \text{ con} \\
& \quad \quad \quad [\text{dis: } i_{prime} \in f_{from}(i)] \text{typ}(i_{prime}) .
\end{align*}
\]

This system of equations can readily be solved by a conventional 'workpile' method. We begin with a 'forward only' pass in which all ivariables other than constants and ivariables for which auxiliary declarations are supplied are initialized to the 'minimum' type \( t_z \); during this pass, the simplified relationships
\( \text{(3a)} \quad \text{typ}(o) = \text{forward}(o) \)

and

\( \text{(3b)} \quad \text{typ}(i) = [\text{dis: } o \in \text{ud}(i)] \text{typ}(o) \)

are used. At the beginning of the second pass, we initialize our workpile to the set

\( \{<\text{backt},i>, i \in \text{ivars}\} + \{<\text{ffrm},i>, i \in \text{ivars}\mid \text{in} \in \text{bfromexit}\} . \)

Here, \( \text{ivars} \) is the set of all ivariables of our program. Then we process the workpile elements. To process an element \(<\text{backt},i>\), we reduce \( \text{typ}(i) \) to \( \text{typ}(i) \text{ con backtype}(i) \); to process \(<\text{ffrm},i>\), we reduce \( \text{typ}(i) \) to \( \text{typ}(i) \text{ con [dis: ip} \in \text{ffrom}(i)] \text{typ}(ip) \). Elements \(<\text{frmo},i>\) and \(<\text{frmi},o>\) can also appear on the workpile. To process \(<\text{frmo},i>\), we reduce \( \text{typ}(i) \) to \( \text{typ}(i) \text{ con [dis: o} \in \text{ud}(i)] \text{typ}(o) \); to process \(<\text{frmi},o>\), we reduce \( \text{typ}(o) \) to \( \text{typ}(o) \text{ con forward}(o) \). Whenever \( \text{typ}(o) \) changes, we put \(<\text{backt},i>\) on the workpile for each argument ivariable \( i \) of \( o \), and put \(<\text{frmo},i>\) on the workpile for each \( i \in \text{du}(o) \). Whenever \( \text{typ}(i) \) changes, we put \(<\text{ffrm},ii>\) on the workpile for each \( ii \in \text{bfrom}(i) \) (actually, it is better to ignore those \( ii \) which belong to \( \text{bfromexit} \)) and put \(<\text{frmi},o>\) on the workpile, where \( o \) is the ovariable to which \( i \) is argument.

2. Calculation of \text{ffrom}, \text{bfro}\text{m}, and \text{bfro}\text{mexit}.

Interprocedural considerations.

As compared to the corresponding approach to the exploitation of 'backwards' type relations outlined in TT, the technique outlined in the preceding pages has the advantage of being 'flow free', and hence adaptable without particular difficulty to interprocedural use. To calculate \text{ffrom}, \text{bfro}\text{m}, and \text{bfro}\text{mexit} we adopt the technique used to calculate \text{ud} and \text{du}. It is convenient to introduce a dummy variable \( \delta \) and insert a dummy argument to \( \delta \) at each program exit, and to allow the set
ffrom(oi) to include both ovariable and ivariable occurrences of oi. Then bfrom is essentially the inverse of ffrom, and bfrom(exit) is \([+: o \in \text{ovars}] \text{bfrom}(o)\), where \(\text{ovars}\) is the set of all ovariables (including the dummy \(\delta\)) of our program.

Thus only ffrom need be calculated. To calculate ffrom(oi), we make use of an auxiliary function reaches(b), which tells us which ovariable and ivariable occurrences of any variable \(v\) can reach the entrance to a block \(b\) along a path free of occurrences to \(b\). Once reaches(b) is available, ffrom(i) can be calculated in a fairly evident way. The basic equation for the calculation of reaches(b) is

\[
\text{reaches}(b) = [+: p \in \text{pred}(b)] (\text{reaches}(p) \ast \text{thru}(p) + \text{occurrences}(p)),
\]

where \(\text{pred}(b)\) is the set of predecessor blocks of \(b\). Here, \(\text{thru}(p)\) is the collection of all ovariables and ivariables whose corresponding variables do not occur in \(p\), and \(\text{occurrences}(p)\) is the set of all ovariables/ivariables which occur in \(p\) but which are not followed in \(p\) by any ovariable/ivariable occurrence involving the same variable.

The values \(\text{thru}(b)\) and \(\text{occurrences}(b)\) are calculated much in the manner explained in Newsletter 134, p. 11.

Much as in NL 134, we must ascribe functions \(\text{thru}(sr)\) and \(\text{occurrences}(sr)\) to each subprocedure \(sr\). Then \(\text{thru}(b)\) is calculated as the intersection of the sets \(\text{thru}(x)\) associated with each of the individual statements \(x\) of \(p\). If \(x\) is a statement other than a function or subprocedure call, then \(\text{thru}(x)\) consists of all ivariables/ovariables whose variables do not occur in \(x\). If \(x\) is a call to a subprocedure \(sr\), then \(\text{thru}(x)\) consists of all ivariables/ovariables which belong to \(\text{thru}(sr)\). If \(x\) is a call to a subprocedure which is somewhat indeterminate and might be either \(sr_1, sr_2, \ldots\), then \(\text{thru}(x)\) consists of all ivariables/ovariables which belong to \(\text{thru}(sr_j)\) for some \(j\). Related rules, which we leave it to the reader to elaborate, hold in calculating \(\text{occurrences}(x)\).
To calculate \( \text{thru}(sr) \) for a subprocedure \( sr \), we prefix the entry block of \( sr \) by a dummy code block which makes an assignment to each global variable referenced in \( sr \) and each parameter in \( sr \). Denote the set of ovariables corresponding to these assignments by \( \text{EXOV} \), and let \( \text{returnstats} \) be the set of all return statements in \( sr \). Then \( \text{thru}(sr) \) and \( \text{occurrences}(sr) \) are equal to

\[
(6a) \quad (+: b \in \text{returnstats}) \text{ reaches}(b) \ast \text{EXOV}
\]
and

\[
(6b) \quad (+: b \in \text{returnstats}) \text{ reaches}(b) - \text{EXOV}
\]
respectively.