SETL Newsletter 140 J. Schwartz

Use-use Chaining as a Technique in Typefinding Sept. 27, 1974

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* the union over v of the set 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 ffrom and bfrom 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 oi is an ivariable occurrence of a variable v, then the type *typ(oi)* associated with oi is the conjunction of:

Ai: the types associated with all ovariables which can supply the value of oi; and:

Aii: the type *backtype(oi)* determined by the manner in which oi is used. This type is a function both of the operation *op* applied to oi and the type information available for the output variable of *op*, and:

Aiii. if oi belongs to bfromexit, then nil; else the disjunction of the types associated with all the elements of ffrom(oi).

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

These relationships are summarized in the following equations:

(2a) for ovariables: typ(o) = forward(o);

(2b) for ivariables:

if i ∈ bfromexit then
[<u>dis</u>: o ∈ ud(i)] typ(o) <u>con</u> backtype(i)
else[<u>dis</u>: o∈ ud(i)] typ(o) <u>con</u> backtyp(i) <u>con</u>
[<u>dis</u>: iprime∈ ffrom(i)] typ(iprime) .

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 tz; during this pass, the simplified relationships

(3a) typ(o) = forward(o)

and

(3b) typ(i) = [dis: o∈ud(i)] typ(o)

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

(4) {<backt,i>, i∈ivars} + {<ffrm,i>, i∈ivars|in ∈ bfromexit} .

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

2. Calculation of *ffrom, bfrom,* and *bfromexit*. Interprocedural considerations.

As compared to the corresponding approach to the exploitation of 'backwards' type relations outlined in TT, the technique outlined in the prceding pages has the advanrage of being 'flow free', and hence adaptable without particular difficulty to interprocedural use. To calculate *ffrom, bfrom,* and *bfromexit* we adopt the technique used to calculate ud and du. It is conveninent to introduce a dummy variable δ and insert a dummy argument to δ 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 bfromexit is [+: o ∈ ovars] bfrom(o), where *ovars* is the set of all ovariables (including the dummy δ) 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

(5) reaches(b) = [+: p ∈ pred(b)] (reaches(p) * thru(p)+ + occurrences(p)) ,

where *pred(b)* is the set of predecessor blocks of b. Here, *thru(p)* is the collection of all ovariables and ivariables whose corresponding variables do not occur in p, and *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 thru(b) and occurrences(b) are calculated much in the manner explained in Newsletter 134, p. 11. Much as in NL 134, we must ascribe functions thru(sr) and occurrences(sr) to each subprocedure sr. Then thru(b) is calculated as the intersection of the sets 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 thru(x) consists of all ivariables/ovariables whose variables do not occur in x. If x is a call to a subprocedure sr, then thru(x) consists of all ivariables/ovariables which belong to thru(sr). If x is a call to a subprocedure which is somewhat indeterminate and might be either $sr_1, sr_2, \ldots$, then thru(x) consists of all ivariables/ovariables which belong to thru(sr_j) for some j. Related rules, which we leave it to the reader to elaborate, hold in calculating occurrences(x).

To calculate 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 EXOV, and let *returnstats* be the set of all return statements in sr. Then thru(sr) and occurrences(sr) are equal to

(6a) [+: b ∈ returnstats] reaches(b) * EXOV

and

(6b) [+: b ∈ returnstats] reaches(b) - EXOV

respectively.