SETL Newsletter # 130A

J. Schwartz
August 1, 1974

Estimates from below of the domain of a mapping

In optimising SETL programs, it can sometimes be useful to know that the domain hd [f] of a mapping f includes all of a set s. This information, if available, can be used either to ensure that a particular value f(x) is not Ω; or to justfy the association of a field for the value f(x) with each element x in s; or, in connection with some future optimiser capable of carrying out sophisticated mathematical transformations, as part, e.g. of a proof that f is a 1-1 transformation defined on s. Even though these potential applications may not yet justify including a mechanism for proving propositions $s \subseteq$ hd[f] in a practial SETL optimiser system, we shall in the present newsletter discuss techniques for establishing propositions of this form. Our discussion will also address an interesting technical question having to do with the treatment of set-theoretical iterators ($\forall x \in s$), and will bring us into implicit but suggestive contact with a fundamental question of optimiser design: how broad a range of issues and methods can be brought into an optimiser without reducing its efficiency to a point at which the optimisations it attains do not repay the cost of optimising?

The constructions which are normally used to build up a function f defined everywhere on a set s are as follows:

i. The set s may be defined first, and then f may be defined for each element of s, either by an iterative loop, e.g.

```
(1)                  f = nℓ; (∀x∈s) f(x) = 0;;
```

or by a set expression, e.g.

(2) $f = \{\langle x,\Omega\rangle,\ x \in s\};$

ii. The set s may be built up progressively, and f defined for the elements of s as they are added to s, e.g. in the pattern

(3) ..., x <u>in</u> s; ...; f(x) = y; ...

Loops (1), as well as the loops implicit in constructions like (2), can be handled as follows. For each loop

(4) $(\forall x \in s)$ *block*;;

generate two auxilary set names s_1, and treat (4) as if it were the following *while*-loop:

```
(5)         s_1 = nℓ; s_2 = nℓ;
            (while s_1 ne s)
                x = ∋(s-s_1);
                s_1 = s_1 with x;
                block;
                s_2 = s_2 with x; oralternatively s_2 = s_1;
            end while;
            s = s_1; oralternatively noop;
```

Here the underlined keyword <u>oralternatively</u> represents a 'binary operator on statements', used to indicate that either of its two 'arguments' (which are both statements) can be applied at a given point since both certainly have the same result.

Note that 'optional' statements, indicated in NL 130 by the use of parentheses, i.e. by (*statement;*), can be written in the form

```
(6)            statement; oralternatively noop;
```

using the statement operator <u>oralternatively</u>. If we treat the loop (1) according to the general schema (5) it becomes

```
(7)                 $s_1$ = nℓ; $s_2$ = nℓ;
                    (while $s_1$ ne s)
                         x = ∋ (s - $s_1$);
                         $s_1$ = $s_1$ with x;
                         f(x) = 0;
                         $s_2$ = $s_2$ with x; oralternatively $s_2$ = $s_1$;
                    end while;
                    s = $s_1$; oralternatively noop;
```

Within the loop we always have <u>hd</u> [f] $\subseteq s_1$ and $s_2 \subseteq$ <u>hd</u>[f], making it plain that <u>hd</u>[f] = s on exit from the loop. Note that a fact - gathering algorithm like that sketched in NL 130 will duplicate this reasoning if only it allows sets like <u>hd</u>[f] to enter freely into the relationships which it manipulates.

Now consider the alternative definition (2) of the mapping f. When schematised and treated *a la* (5), it will become

```
f = nℓ;
$s_1$ = nℓ; $s_2$ = nℓ;
(while $s_1$ ne s)
        x = $\ni$(s - $s_1$);
        $s_1$ = $s_1$ with x;
        t = <x,0>;
        f = f with t;
        $s_2$ = $s_2$ with x; oralternatively $s_2$ = $s_1$;
end while;
s = $s_1$; oralternatively noop;
```

Within the loop we always have $\underline{hd}[f] \subseteq s_1$, and our optimiser will know this since it will be aware of the fact $t(1) \in s_1$. Similarly, the optimiser will know that $s_2 \subseteq \underline{hd}[f]$. However it is more complicated to prove using (8) that f is single-valued than it is to prove the same fact by analysing the code (7). The proof is as follows: since $\underline{hd}[f] \subseteq s_1$, we have $x\ \underline{n}\in \underline{hd}[f]$ at the start of (8). Thus $\underline{hd}[\{t\}]$ is disjoint from $\underline{hd}[f]$, which guarantees that f remains single-valued. This argument also will be within range of a fact gatherer like that of NL 130, if only free use of sets like $\underline{hd}[f]$ is allowed.

If s and f are built up progressively in the manner illustrated by (3) then a proof that $s \subseteq \underline{hd}[f]$ can be obtained along the following lines: before x $\underline{in}$ s is executed, $s \subseteq \underline{hd}[f]$ should be true; and then after $f(x) = y$ is executed $s \subseteq \underline{hd}[f]$ will be true again. For this line of reasoning to be accessible to an optimiser, it will clearly have to manipulate at least some limited class of propositions of the form $s \subseteq \{x\} + \underline{hd}[f]$.

We see in sum that to allow automatic recovery of all the deductions described in the last few pages, it is sufficient to make the following additions to the fact-gathering algorithms described in Newsletter 130:

i. If f is a set of ordered pairs used as a mapping, its domain *dom* = $\underline{\text{hd}}$[f] should be regarded as a set able to enter into relationships *x* R *dom* and *dom* R *x*. This can be done by generating a variable to represent *dom*, and inserting update-code to modify *dom* whenever f is is modified. Note that f = $\underline{n\ell}$ makes *dom* = $\underline{n\ell}$; f = s makes dom = $\underline{\text{hd}}$[s]= dom_s, f = f + g makes dom = dom + $\underline{\text{hd}}$[g], etc. An indexed assignment f(x) = y makes dom = dom + {x}, if y $\underline{\text{ne}}$ Ω (which might well be known from prior typefinding) and dom = dom - {x} if y $\underline{\text{eq}}$ Ω. If y may or may not be Ω, one must use dom = dom $\underline{+}$ {x}, where $\underline{+}$ is a new SETL primitive with semantics which should be clear to the reader. The plausibility and necessity rules which apply to relationships involving *dom* are the same as those which apply to any other set, with one exception in the necessity rules, which exeception we will explain immediately below.

ii. If f and x appear together in the context f(x) = ..., then we apply the following procedure:

(a) Find all statements of the form x $\underline{\text{in}}$ s, s = s $\underline{\text{with}}$ x, and s = s + {x}, and prefix each such statement with an auxiliary assignment x = x. Convert the forms s = s + {x} and x $\underline{\text{in}}$ s to the form s = s $\underline{\text{with}}$ x. Then perform data-flow analysis.

(b) If the only ovariable chained to the ivariable x of f(x) = ... is the ovariable x of one of the assignments x = x thus introduced, allow statements

(9) $$o \subseteq \{x\} + o_1,$$

where o is any ovariable occurence if s, and where o_1 is any ovariable occurence of *dom* = hd[f], to appear among the hypotheses being processed. In (9), which we shall prefer to write as

(9') $$o \subseteq_x o_1,$$

x is the target ovariable of some particular one of the assignments x = x introduced by rule (a).

(c) The plausibility and necessity rules for propositions (9') include those which apply to ordinary assignments $o \subseteq o_1$. However, for (9') to be plausible we also demand that $x \in o$ and $x \in_1 o_1$ be plausible; and at statements of the form $o = i_1$ with i hypothesis (9') will be confirmed if the hypothesis $i \subseteq o_1$ is available and if x is the only ovariable chained to i.

(d) Apply the following modified necessity rule at every assignment f(i) = y for which the only ovariable chained to the ivariable i is the x of (9'): if the hypotheses y ne Ω and $o \subseteq_x dom_f$ are available before this assignment, then $o \subseteq dom_f$ is available after it.

With these modifications, the procedure outlined in NL 130 can be used to prove that a map f is defined everywhere on a set s; and also to prove that f is single-valued.

7. A general observation concerning 'source' and 'schematized' forms of code in optimisation.

The following general remark is suggested by the technique used in section 1 above to treat set iterators ($\forall x \in s$).

When source code is schematised to prepare it for analysis, expressions and other relatively 'integral' constructs easily detected in the source will expand into longer,less integral sequences, e.g. of quadruples.

Even though sound global optimisation techniques will withstand most of the impact of this expansion, information will occasionally be lost, at least in the sense of being hidden in an implicit form from which its explicit recovery by global analysis would be over-expensive. To avoid or minimise this difficulty, one will of course want to choose schematisation rules rather carefully. In particular, it can be useful to examine the source code for optimisation-significant local features while its schematised form is being generated, and to vary the schematisation, perhaps by compiling in extra indicators, in a way reflecting any significant source-level features which are detected. The <u>oralternatively</u> construction used in section 1 is handy for this purpose, as it makes a logical 'and' operations available for use during the global analysis which follows schematisation; note that in this global analysis converging flow paths are associated not with the 'and' operation but with the logical 'or' operation.