SETL Newsletter No. 205 — Micha Sharir

On Disjointness Detection in Automatic Data-Structure Choice

July 11, 1978

In this newsletter we suggest an algorithm to detect disjointness among SETL program objects. This algorithm can be applied toward detection of single-valued maps and detection of cases in which sets can be represented as simple linked lists (without hashing support). The algorithm that we suggest here resembles in some points Schwartz' approach toward detection of inclusion and membership relationships between SETL program objects, but due to its limited goals, our approach tends to be somewhat simpler and easier to implement.

We start by giving some rules indicating when a linked list representation for sets and tuples can be expected to be an efficient one. These rules will motivate the algorithm to be proposed later.

Let t be a homogeneous tuple in a SETL program. t is appropriately represented as a linked list if it is only subject to the following Q1 operations: ADD, WITH, ASN, NEXTD, and to component retrieval and storage operations of the form t(1), t(#t), t(#t+1) := . (We assume that the last element in the list structure can be accessed rapidly via a pointer from the first element.) The preceeding criterion can be checked easily simply by scanning all occurrences of t in the program. (Note that the construct t(#t), e.g., can be easily detected if the code to be analyzed makes use of the present unique naming scheme for temporaries.) Thus, finding tuple candidates for a linked-list representation is a relatively simple task, which involves no particular problems.

The situation is more complex if we consider sets instead of tuples. Let S be a set in a SETL program. S is appropriately represented as a linked list if it is only subject to the following Q1 operations:

ARB, FROM, NEXT, ASN,
S <u>with</u> x, where x is known not to belong to S just before any execution of this instruction,
S + T, where S and T are known to be disjoint and T is also to be represented as a list.

In this set case we see that to detect such a representation we need some disjointness information concerning sets and their potential elements. Interestingly enough, similar disjointness information can be used to detect single-valued maps. Specifically, a map f is single-valued if it appears only in the following contexts:

f := g, where g is known to be single valued (typically g is the null map),
f(x) :=
as an argument to the operations DOMAIN, RANGE, NELT, OF, etc.,
f <u>with</u> [x,y], where <u>domain</u> f and x are known to be disjoint just before any execution of this instruction.

It follows that disjointness detection analysis can aid in selecting appropriate data structures. Specifically we suggest that such an analysis be performed only after the name-splitting phase, as it is more convenient to assume that all program objects have unique type and representation (which is the case after the name splitting phase). The algorithm to be described below is therefore intended to be part of the refinement phase of the automatic data structure choice, in which the basic relationships between program objects and their representation in terms of program bases have already been established, and in which it remains to decide which kind of based representation (local, remote, sparse or list) is most efficient for individual sets and maps.

An additional advantage of performing our algorithm at this stage is that we can restrict disjointness detection to sets, map domains and elements known to be based on the same base. This will decrease significantly the number of possible candidates for disjointness and will make our algorithm quite a practical one.

Our algorithm is a kind of global data-flow analysis. It will be described simply by defining the data-flow framework involved, from which the precise details of implementation can be worked out in a routine manner, using any convenient data-flow analysis technique.

We first define a semilattice L of possible data values to be collected during our analysis. Each element A of L is a set of unordered pairs. A pair $\{S_1,S_2\}$ is in A iff S_1 and S_2 denote two sets appearing in a SETL program which are based on the same base and are known to be disjoint. The meet in L is set intersection. To incorporate elements and map domains into our disjointness scheme, we also maintain two abstract maps: SETOF, which maps each element x of a base to an atom designating the singleton set {x}, and DOMOF, which maps each map f which is domain-based on some base to an atom designating the domain set of f. Thus, for example, if t is a pair [u,v], then DOMOF(SETOF(t)) designates the set {u}. Also, if {SETOF(x),S} $\in$ A $\in$ L, then A indicates that x $\notin$ S. Similarly, if {SETOF(x),DOMOF(f)} $\in$ A then A indicates that x $\notin$ domain f.

The set F of data-propagation maps of our framework is best described by first defining, for each program instruction I, a map f_I which describes the information change effected by execution of I. F can then be constructed by appropriate compositions and meets of these elementary maps.

We will describe below f_I for several typical SETL instructions I. Let A $\in$ L denote information known just before executing I. Then:

(1) I: S := <u>nℓ</u>; for a based set S. f_I(A) is obtained by adding to A all pairs {S,T}, where T is a set based on the same base B of S, or T = SETOF(x), where x is an element of B, or T = DOMOF(f), where f is a map which is domain-based on B, or T = DOMOF(SETOF(p)), where p is a pair whose first component is an element of B. (We omit in this description details pertaining to more efficient implementation of this algorithm, such as possibly maintaining sets known to be null separately in a disjointness data, thereby expediting the application of f_I(A) for the above I.)

(2) I: f := <u>nℓ</u>; for a domain-based map f. Proceed as in (1), using DOMOF(f) instead of S.

(3) I: x <u>from</u> S; f_I(A) is obtained by deleting all pairs {SETOF(x),T} from A, and then adding the pair {SETOF(x),S} as well as all the pairs {SETOF(x),T}, where {S,T} $\in$ A.

(4) I: x := <u>next</u> S; (i.e. iteration over a based set S). Here

we make use of the semantic translation of iterations, in which S is first assigned to a 'shadow' set S_1 and then iteration is performed on S_1. We can thus interpret I as 'x <u>from</u> S_1' and act as in (3) above.

(5) I: x := <u>arb</u> S; Proceed as in (3), only do not add {SETOF(x),S} to A.

(6) I: S <u>with</u> x; Remove from A {SETOF(x),S}; leave pairs {S,T} in A only if {SETOF(x),T} is also in A.

(7) I: S <u>less</u> x; Add to A the pair {SETOF(x),S}.

(8) I: S := $\{x_1,x_2,\ldots,x_n\}$; Interpret this as the sequence S := <u>nℓ</u>; S <u>with</u> x_1; ... S <u>with</u> x_n;

(9) I: S := $S_1 + S_2$; Delete all pairs {S,T} from A; add pairs {S,T} to A for which both $\{S_1,T\}$ and $\{S_2,T\}$ are in A.

(10) I: S := $S_1 - S_2$; Delete all pairs {S,T} from A; add pairs {S,T} to A if $\{S_1,T\} \in A$.

(11) I: S := $S_1 * S_2$; Delete all pairs {S,T} from A; add {S,T} to A if either $\{S_1,T\} \in A$ or $\{S_2,T\} \in A$. However, if $\{S_1,S_2\} \in A$, interpret I as S := <u>nℓ</u>;

(12) I: f(x) := y; Interpret as 'DOMOF(f) <u>with</u> x', using (6).

(13) I: f{x} <u>with</u> y; Same as (12).

(14) I: f lessf x; Interpret as 'DOMOF(f) <u>less</u> x', using (7).

(15) I: S := <u>domain</u> f; Interpret as 'S := DOMOF(f)', using (19) below.

(16) I: f <u>with</u> t; for a domain-based map f. Since f is a map, t must be a pair. Interpret I as

DOMOF(f) := DOMOF(f) + DOMOF(SETOF(t));

This calls for proper handling of instructions manipulating pairs. For example:

(17) I: t := [x,y]; Interpret as 'DOMOF(SETOF(t)) := {x}'.

(18) If I is a retrieval of a pair t from a domain-based map f, then interpret I as if DOMOF(SETOF(t)) has been retrieved (as a

subset) from DOMOF(f), using (3), (4) or (5) with the above sets replacing SETOF(x) and S respectively.

(19) I: x := y;

(a) If x,y are based sets, remove all pairs {x,T} from A; add pairs {x,T} to A if $T \neq x$ and $\{T,y\} \in A$.
(b) If x,y are domain-based maps, interpret as 'DOMOF(x) := DOMOF(y)', using (a) above.
(c) If x,y are pairs whose first components are base pointers, interpret as 'DOMOF(SETOF(x)) := DOMOF(SETOF(y))'.
(d) If x,y are base pointers, interpret as 'SETOF(x) := SETOF(y)'.

(20) Our scheme excludes (split) variables which are subject to arithmetic operations, are read in, etc. Each conversion from an unbased split variable to a based one is considered as a creation of a new value for the based variable, which removes any disjointness information concerning this variable from A.

These rules summarize the effect on L of execution of the most common SETL instructions having based arguments. We have thus defined a data-flow framework (L,F). It is not distributive, as the following example indicates: Let $A_1 = \{\{S,T\}\}$, $A_2 = \{\{S,U\}\} \in L$, and let I: V := T * U; Then both $f_I(A_1)$ and $f_I(A_2)$ contain {V,S}, but $f_I(A_1 \wedge A_2)$ does not. Nevertheless, our framework is defined for any flow-graph (or program), and yields a data-flow problem solvable by any convenient standard technique.

Let us demonstrate the way in which our algorithm applies to several typical examples:

Example 1: Consider the following expansion of the set-former

$$T := \{x \in S \mid c(x)\}$$

```
I1    T := nℓ;
I2    S1 := S;
I3    (∀x ∈ S1)
I4        if c(x) then
I5            T with x;
```

```
I6          end if;
I7    end ∀;
```

Let A_I denote the data known just before executing an instruction I. We assume that there is one underlying base B such that T,S,S_1 and Sx ≡ SETOF(x) are all sets based on B. If we propagate information in the order of the code, we obtain, after each propagation step:

$A_{I_1} = \emptyset$

$A_{I_2} = \{\{T,S\},\{T,S_1\},\{T,Sx\}\}$, using rule (1).

$A_{I_3} = A_{I_2}$ ($\{T,S_1\}$ is removed and then added back, by rule (19))

$A_{I_4} = \{\{T,S\},\{T,S_1\},\{Sx,S_1\},\{Sx,T\}\}$

(remove $\{T,Sx\}$ and then add it back with the pair $\{Sx,S_1\}$, using rule (4))

$A_{I_5} = A_{I_4}$ (I_4 does not change any value)

$A_{I_6} = \{\{T,S_1\},\{Sx,S_1\}\}$

(the pair $\{T,Sx\}$ is removed, and from all the other pairs containing T, only $\{T,S_1\}$ is left, since $\{Sx,S_1\}$ is also in A, by rule (6))

$A_{I_7} = A_{I_4} \wedge A_{I_6} = A_{I_6}$

$A_{I_3} = A_{I_3} \wedge A_{I_7} = \{\{T,S_1\}\}$

$A_{I_4} = \{\{T,S_1\},\{Sx,S_1\},\{Sx,T\}\}$, using rule (4)

$A_{I_5} = A_{I_4}$

A_{I_6}, A_{I_7} are unchanged

The propagation stabilizes as shown above, yielding the information that T and {x} are disjoint before any insertion of x into T. Thus T can be represented by a linked list (as far as the above code fragment is concerned).

Example 2: Consider the following expansion of the map-former

$$f := \{[x,e(x)] : x \in S\}$$

```
I_1    f := nℓ;
I_2    S_1 := S;
I_3    (∀x ∈ S_1)
I_4        y := e(x);
I_5        t := [x,y];
I_6        f with t;
I_7    end ∀;
```

Again we assume that there is only one underlying base B such that DF ≡ DOMOF(f), S, S_1, Sx ≡ SETOF(x) and DST ≡ DOMOF(SETOF(t)) are all sets based on B. The propagation steps will yield

$A_{I_1} = \emptyset$

$A_{I_2} = \{\{DF,S\},\{DF,S_1\},\{DF,Sx\},\{DF,DST\}\}$, by rule (1)

$A_{I_3} = A_{I_2}$ (as in example 1)

$A_{I_4} = \{\{DF,S\},\{DF,S_1\},\{DF,Sx\},\{DF,DST\},\{Sx,S_1\}\}$

(remove {DF,Sx} and then add it back and add $\{Sx,S_1\}$, by rule (4))

$A_{I_5} = A_{I_4}$ (I_4 is an unbased instruction which does not yield any disjointness information)

$A_{I_6} = \{\{DF,S\},\{DF,S_1\},\{DF,Sx\},\{DF,DST\},\{Sx,S_1\},\{DST,S_1\}\}$

(interpret I_5 as 'DST := Sx', applying rule (18))

$A_{I_7} = \{\{Sx,S_1\},\{DST,S_1\},\{DF,S_1\}\}$

(interpret I_6 as DF := DF + DST, applying rule (16))

$A_{I_3} = A_{I_3} \wedge A_{I_7} = \{\{DF,S_1\}\}$

$A_{I_4} = \{\{DF,S_1\},\{DF,Sx\},\{Sx,S_1\}\}$

$A_{I_5} = A_{I_4}$

$A_{I_6} = \{\{DF,S_1\},\{DF,Sx\},\{Sx,S_1\},\{DST,DF\},\{DST,S_1\}\}$

$A_{I_7} = \{\{DF,S_1\},\{Sx,S_1\},\{DST,S_1\}\}$ (unchanged)

This stabilizes the propagation. We thus obtain that whenever

I_6 is executed, the domain of f and {x} are disjoint, so that f is a single-valued map.

<u>Remarks</u>: (1) Note that our algorithm treats potential disjoint sets as sets of abstract elements (base pointers) for which no special properties are assumed or explored. Thus, e.g., our algorithm will not detect the fact that the set $\{x + 1 : x \in S\}$ can be constructed as a list, a property which can be deduced only if we use the fact that addition of a constant integer is a one-one operation. Thus any disjointness information that our algorithm can collect is based only on assignment, retrieval and embedding operations.

(2) Our algorithm is much more modest than the approaches set by Schwartz and Tsui, which aim at detection of various inclusion and membership relations, of which disjointness is only one special case. Their methods are indeed more powerful. For example, consider the transitive-closure code given in Schwartz's "Optimization of Very High Level Languages II". Our algorithm will not detect the fact that 'new' can be represented as a linked list, a fact that can be deduced only by noting that new $\subseteq$ all throughout the code, a property which we do not aim to detect. However, their algorithms (especially Schwartz') are much more complicated, may require several pre-processings of the code, including insertion of dummy instructions, and are much more space-consuming. We believe that the algorithm suggested above is a reasonably implementable one, which will detect most of the common single-valued maps and possible linked list representations.