«LE MATEMATICHE»
Vol. XLIII (1988) - Fasc. I, pp. 53-78

SET ORIENTED LANGUAGES AND PROGRAM TRANSFORMATIONS

PHILIPPE FACON (Evry) - YO KELLER (Paris) (*)

Set constructs and notations provide in many areas an unprecedented expressive power. Sets are nevertheless almost non-existent in most programming languages since they don't have a general-purpose efficient enough representation. Only global transformations, taking into account the context of set constructs and operations may provide a reasonable efficiency. After a brief survey of existing Set Oriented Languages, we present recent developments taking place at New York University concerning SETL and its successors, especially fixed-point specifications, elimination of repetitive evaluations by finite differencing and elimination of associative access costs by an appropriate Data Structure Selection for implementing sets. In this framework we present two original contributions: the first one concerns a rewrite operator on sets for dealing with fixed points of some non-monotonic transformations. The second one concerns Data Structure Selection and how we have extended and reformulated its mechanism as a special kind of type inference, relatively easily implemented in Prolog.

(*) Part of this work has been supported by the Esprit project Sed, and in conjunction with NYU-Computer Science Department, under contract with Thomson-CSF/DSE.

1. The need for set theoretic dictions in programming.

The classical set theory is a formalism which is quite widespread and close to the intuition, at least in its naive forms. Its expressive power is especially significative in the following kinds of applications:

* applications dealing with *structured information*, as in business data processing, or in CAD-CAM. In such applications many objects are readily modeled by set constructs: the set of employees in a given department, the set of activities of an individual, the mapping of a set of attributes to an individual identification, and its converse, the set of components of a mechanical device, and recursively the components of each of these components, etc.

The main data model, the relational model, does not allow the direct expression of set attributes: instead, this notion is translated as a kind of dependency between atomic values in tuples. Such dependencies are neither easily expressed by a user nor easily analyzed by the system. This is why recent developments in data processing languages for «complex objects» [Abi, 87] are essentially based upon set primitives, whether in an algebraic framework (the language is a set of composable operators) or in a predicative one (the language is based upon 1st order predicate logic). Let us note that these languages are exclusively concerned with the data processing aspects, and that the atomic values are not interpreted (e.g. their type is not analyzed), only equality tests are available. Such languages will doubtless play an important role with the coming of multimedia data bases, storing all types of information including text, sound and images.

* *algorithm design*, where the possibility of defining freely sets and of applying to them global operations lead to particularly elegant, i.e. precise and concise, formulations of many problems. For instance a graph G with vertices $\{a, b, c, d\}$ and edges $[a, b]$, $[a, c]$, $[c, d]$ is described as a set of pairs $G = \{[a, b], [a, c], [c, d]\}$. Then the existence of a cycle in G can be written as:

$$\exists C \in \text{ powerset } (G) | \forall X \in C, \exists Y \in C : X[2] = Y[1]$$

This expression is directly executable in SETL. Other simple and useful examples of set expressions (all are actually executable in SETL) are:

$\forall p \in [2..n-1] \| n \bmod p \neq 0$	prime(n)
$\{[x, y] : [y, x] \in f\}$	f^{-1}
$\{[x, z] : [x, y] \in R,\ z \in S\{y\}\}$	$S \circ R$
$\{[x, y] : x \in A,\ y \in B\}$	$A \times B$
type (f) = set and $\forall p \in f \|$ type (p) = tuple and $p(1) \neq om$ and $\#p = 2$	map (f) (f is a 2-ary relation)
map (f) and $\forall x \in$ domain $f \| \#f\{x\} = 1$	smap (f) (single-valued map)
smap (f) and $\forall x \in$ domain f, $\forall y \in$ domain f:$x \neq y \Rightarrow f(x) \neq f(y)$	oneone (f) (injective function)
$\{f \in$ powerset $(A \times B) \|$ domain $f = A$ and smap $(f)\}$	functions from A to B

Similarly all usual data structures, e.g. linked list, stack, queue, tree, multiset, polynomial have simple abstract presentations in terms of set constructs and operations.

* *semantic analysis* of programs, both deterministic and non-deterministic programs. The basic data structures are data and control flow graphs which are well represented as (multivalued-) maps. The basic queries concern the propagation pathways of variables – and expressions – values and types: where, in a program, is the value defined, where is it used? Likewise, one has to consider sets of possible values at execution time for specific program parameters (e.g. the type of an expression). Most properties may then be formalized as fixed-points for functions on sets. Again the analogy with databases is striking. As observed in [Aho, 79]: *«fixed-point operations arise naturally in a variety of database applications. In an airline reservation system, for example, one may wish to determine the number of possible flights between two cities during a given time period. [...] No such query can be couched in relational algebra»*. The computation of

fixed-points requires that sets and relations be 1st-class objects, in order to express properties like being monotone:

$$s \subseteq t \Rightarrow f(s) \subseteq f(t)$$

or to construct a procedure to compute such a fixed point, e.g. as a while loop:

```
$s := \{\}$;
(while $g(s) \neq \{\}$)
    $s := s \cup g(s)$;
end;
```

2. Set primitives in programming languages.

There is a priori, many different ways of integrating sets to a programming language. One has to design first the ur-elements (primitive elements which are not sets): it is still possible to exclude such ur-elements and reconstruct everything starting with the empty set; however this seems a rather futile exercise nowadays in computer science. The ur-elements will denote elements the structure of which is ignored. One has then to determine which operations will be used to create and process sets. For instance infinite sets may be made available. Otherwise, it is possible to prevent them syntactically by restricting definitions of sets in comprehension. The freedom to nest set constructs and quantifiers may be also restricted.

An extreme case is that of Pascal: finite (very small!!) sets, without imbrications nor quantifiers or definitions in comprehension. In fact, with the exception of the very recent above-mentioned data-base studies, there are very few languages implementing a significant part of set theory. Among imperative languages SETL is a – relatively old – example offering a quite complete set of primitives for handling hereditarily finite sets.

In Logic Programming the need for set expression emerged rapidly, especially for expressing the set of terms satisfying a given predicate. Indeed the simulations of sets with lists (e.g. accumulators of single-question solutions) are difficult to implement in full generality and inefficient. It should be stressed that other 2nd order possibilities (variable predicates, λ-expressions) are much easier to simulate by meta-programming. This is why recent version of Prolog offer some direct set processing primitives (set-of, find-all), extending the First Order Logic and usually praised by programmers.

Among functional languages, some like Miranda or Me Too [Naf, 86] allow some set oriented operations, including with infinite sets (by means of a lazy evaluation). However, no more than in Prolog, sets may be considered truly as native language objects. An exception could be S3L [Lac, 88] under development at the University of Orleans, which considers itself as based upon (a) set theory.

Thus, despite its advantages, set theory has had, until now only relatively little success in programming languages. The main reason seems to be an efficiency problem: there are no general representation of sets able to support efficiently all the set operations. Before examining possible solutions to this problem, we will present rapidly the framework in which these solutions have been worked out, i.e. SETL and derived languages.

3. Set primitives in SETL and its successors.

Developed at NYU some 15 years ago, SETL provides a complete set of primitives for hereditarily finite sets. Besides sets, the basic language structures are the tuples and the maps: the latter one being simultaneously sets of pairs, graphs of functions or binary relations, the former ones being plain unlimited lists. The following table provide a quick overview of the extent and flexibility of the SETL primitives (the table includes the two SQ+ fixpoint operators(1)).

(1) The SQ+ language exposed in 4 is a functional language essentially equivalent to the SETL expression language presented here.

SETL set, maps and truple expressions

$s \cup t, s \cap t, s - t$	set union, intersection, difference
$s \subseteq t, s = t$	subset and equality tests
$\#s$	set cardinality, sequence size
$\ni s$	arbitrary choice
$x \in s, x \notin s$	membership tests
$y \in g\{x\}$	test $[x, y] \in g$
$\{\}, \{x1, x2, \ldots, xn\}$	empty set, enumerated set,
$[], [x1, x2, \ldots, xn]$	empty tuple, enumerated tuple
$\{e(x) : x \in S \| k(x)\}$	set former
$[e(x) : x \in s \| k(x)]$	tuple former
$\forall x \in s \| k(x), \exists x \in s \| k(x)$	boolean valued quantifiers
domain f, range f	domain and range of a map
$f(x), t(i)$	image of an element by a map (undefined if not unique), i-th element of a tuple
$f\{x\}, f[x]$	set of images of x, set of images of elements of x
$lfp(s, e), gfp(s, e)$	least (greatest) fixed point of e greater (smaller) than s (specific to SQ+)

SETL main iteration constructs and set, maps and tuple instructions(2)

s with $:= x; s$ less $:= x$	set or tuple element additions and deletions
$g\{x\}$ with $:= y; g\{x\}$ less $:= y$	add (delete) pair $[x, y]$ to (from) map g
$s := \{\}, f(x) := \{\}, f\{x\} := \{\}$	assign empty set to a variable, as the image value of an element in a map, as image set
$f(x) := \ldots, f\{x\} := \{\ldots\}$	assign a value as the image value of an element in a map, as image set
$g(x) := \Omega$	remove x from domain g
(for $x \in s$) [Block] end	execute Block for each element $x \in s$
(while Bool) [Block] end	execute Block as long as condition Bool holds. Bool is any boolean expression (see above)

(2) The SML language is a subset of SETL consisting of the elementary expressions, assignments to sets and maps, set element additions and deletions (there is no global operation between two sets). It is an intermediate language for the translation of SQ+ into imperative and setless low level languages.

The type of SETL objects is extremely dynamic: a variable may change its type, even the same occurrence, during the same execution. Although (type) declarations are possible, they are not mandatory, and most SETL programs are declaration free. SETL is well suited for the rapid development of prototypes. A complete tutorial and presentation of the language may be found in [SchD, 86].

SETL is a rather old language and some of its defects are rather obvious. For instance problems concerning the scope of variables, some side-effects and the semantic of some constructs are being addressed in a new version «SETL2» under development at NYU [Smos, 88]. One could also question the imperative nature of the language, but it seems to fit its vocation as a prototyping language, for prototypes evolving gradually toward an Ada implementation as suggested in [BlG, 88]. One could also question the weak typing, since its disadvantages are well known: type errors go undetected before run-time, performance loss due to continuous run-time type-checking. This is why a new type model for SETL2 is under study. This model, outlined in [Hen, 87], should preserve the flexibility of the language, including it being declaration-free, while enforcing some form of strong typing. This model constitutes an extension of SETL with procedures as first-class objects and tagged values. The model is inspired by ML's polymorphism and type inference.

4. SETL Program Transformations.

Program Transformation has always been an active area of Research in the SETL community. It resulted mainly in two distinct systems: the SETL Optimizer [Fre, 83] and the RAPTS system [Pai, 87]. We will discuss here the latter system. Bob Paige and his team have designed a specification language, *SQ*+[CaiP, 88], and an associated transformational system: RAPTS. The goal is the automatic generation of efficient imperative programs for a certain class of problems: those which could be reduced to the search for a fixed point of a set function monotone (or antimonotone) wrt the set inclusion. Indeed an important number of set oriented specifications can be expressed as $SQ+$ fixed point expressions e.g.:

$$\text{themin } S : X \subseteq S | S = h(S)$$

denoting the smallest set S containing X such that $S = h(S)$. This expression is undefined if there is no unique least fixed point. An inductive definition of a set can always be turned into a fixed-point normal form (this is even also the case of any partial recursive function). For instance the *powerset* of X is:

(3) $$\text{themin } S : \{\{\}\} \subseteq S | S = S \cup \{\{x\} \cup y : x \in X, y \in S\}$$

Let us consider the *attribute closure* problem arising in relational data bases. Let U be a set of attributes. Given a subset X of U and a map f between subsets of U (the pairs in f are the socalled functional dependencies), one has to determine the smallest set Z of attributes such that:

$$X \subseteq Z \text{ and } \forall Y : Y \subseteq Z \Rightarrow f(Y) \subseteq Z$$

Using the domain operator (which applied to a map or 2-ary relation returns the set of all 1st components of the tuples in the relation), Z can be directly expressed in $SQ+$ as:

$$Z = \text{themin} S : X \subseteq S | (\forall Y \in \text{ domain } f | Y \subseteq S \Rightarrow (f(Y) \subseteq S)$$

By means of simple syntactic transformations this specification is tranformed into the fixed-point expression:

(4) $$\text{themin} S : X \subseteq S | S = S \cup f[\{Y \in \text{ domain } f | Y \subseteq S\}]$$

where we use the classical notation

$$f[P] = \{f(x) : x \in P\}$$

The *attribute closure* problem is now well under the form (2).

When h in (2) is monotone (wrt. set inclusion) then Tarsky theorem provides the solution by an iterarion:

```
$X := S;$
loop $X := h(X)$ until converge;
```

i.e., stated with more conventional notations[3]

```
$X := S;$
while $X \neq h(X)$ do
    $X := h(X);$
end;
```

Monotonicity is an undecidable property. However sufficient conditions for being monotone can be stated as syntactic patterns: identification of known tabulated primitives, preservation of monotonicity through function composition, etc. By this method, and with some other minor transformations one obtains a first algorithm for solving the *attribute closure* problem[4]

```
    $S := X;$
    while exists $z \in f[\{Y \in \text{domain } f | Y \subseteq S\}] - S$ do
(5)     $S$ with := $z$;
    end;
```

The complexity of this iteration is $O(n^2)$, n being the size of U, whereas the direct fixed-point search is exponential. The *Finite-Differencing* incremental re-evaluation techniques (exposed later in 6)

[3] In what follows we take some lexical freedom from the actual SETL syntax.

[4] S with:= z is the same as $S := S$ with z; or equivalently: $S := S \cup \{z\}$;

suppress the repetitive evaluation of $f[\{Y \in \text{domain } f | Y \subseteq S\}] - S$ at each step in the iteration. Instead the value computed at the previous iteration step is used and updated. This results in a linear complexity, if one admits that associative accesses (e.g. membership tests) can be made in constant time. This is in theory possible with hashing techniques, but in practice, constant time is not uniformly achieved. Besides space complexity is essential in set oriented programming. Whence the selection of set representation according to an access analysis (see 7, 8) completes this optimization: the initial variables, as well as those introduced by the finite-differencing techniques are represented in such a way as to minimize the cost of associative accesses. The resulting complexity is then truly linear in both space and time.

Two other techniques, more classical, improve execution-time (by a constant factor): loop fusion and dead code elimination. These optimizations clean some of the useless or redundant code introduced by the transformational technique during the intermediate phases of code generation.

Together all these techniques enable the RAPTS system to generate many efficient algorithms from their specification. This has been illustrated with semantic analysis algorithms (flow-graph analysis) and with difficult graph algorithms, such as Hopcroft-Tarjan's graph Planarity Testing [Cai, 87]. The latest developments around RAPTS concern the resolution of systems of fixed-point equations and the recomputation of fixed-point after the modification of a parameter.

5. A rewrite operator on sets.

In [DaM, 86], Dahlhaus and Makowsky discuss in detail the choice of primitives in languages *à la SETL*. They propose as criteria for the basic primitives: complexity, independence and completeness (wrt. a semantic definition of computable functions over hereditarily finite sets). They present a language satisfying these criteria, a kind of well-defined subset of SETL, with the same expressive power, using operators for: union, complement, pair construction, loop and parallel application on all the elements of a set.

[DaM, 86]'s criteria concern basic operators, i.e, the target language for algorithms generated by fixed-point specifications: The criteria which we retain for higher-level operators – i.e. at the specification language level – are completeness (the ability for expressing everything compactly with a few high-level operators only), correctness provability (in this respect, the *while* operator is far too general) and transformational derivability. This is why we have attempted to generalize the converge iteration

$$\text{loop } X := h(X) \text{ until converge;}$$

to non-monotone functions, i.e. functions which are neither monotone nor anti-monotone. Let us observe first of all, that this form of expression, although imperative, is sometimes more appropriate than the associated fixed-point expression (which is of a higher-level and may hide more information, e.g. arbitrary selections). Let us state informally our rationale: the set X in the iteration is not in general fully modified at each step. The converge loop attempts to capture the corrections being applied successively to the same set leading progressively to the result. It is this mechanism which we propose to generalize. Intuitively, instead of constructing the solution with successive additions to a set, one constructs a succession of local transformations which take us each time nearer to the solution. Element by element transformations being too limited, one will consider as transformations rewriting of part of the set.

More precisely let P and Q be set valued expressions with free variables. All the free variables of Q are also free in P. We consider substitutions to the free variables of only terms without free variables. We write $\sigma(P)$ the result of the substitution σ applied to P.

Then, the elementary rewriting *rewrite* 0 $(P \to Q)$ is applicable to S iff there is a substitution σ that satisfies:

– the matching condition: $\sigma(P) \subseteq S$

– and the change condition: not $(\sigma(P) \subseteq \sigma(Q) \subseteq S)$

I.e. a certain subset of S matches P and its replacement will modify S.

The result of that elementary rewriting is:

$$S - \sigma(P) \cup \sigma(Q)$$

By the change condition, that result is necessarily different from S.

For instance, rewrite $0(X \cup \{\{X, a\}\} \rightarrow X \cup \{a\})$ – where X is a free variable – is applicable to $\{c, d, b, e, \{\{b, c\}, a\}, f\}$ because the substitution $X = \{b, c\}$ and the result is $\{c, d, b, e, a, f\}$.

Then, the full *rewrite* operation repeats that elementary rewriting until it cannot be applied any more: for all part of S, either it doesn't match the pattern P or it violates the change condition.

We can define in SETL:

$$\textit{rewrite}\ (P \rightarrow Q) \text{ in } S$$

to be

```
    while $\exists X \subseteq S | C(X)$ and
        not ($X$ subset $F(X)$) or not ($F(X)$ subset $S$)) do
(7)
        $S := S - X \cup F(X)$;
    end while;
```

where $C(X)$ corresponds to $\exists \sigma | X = \sigma(P)$ and $F(X)$ to $\sigma(Q)$.

Rewrite is non-deterministic: at any given moment, several parts of the set S may satisfy the match and the change conditions. Any one of those parts is selected for rewriting, and this is well reflected by the semantics of the SETL $\exists$ operator.

Let us observe that a monotone iteration on a set S to a fixed-point of a function f may be written:

$$\textit{rewrite}\ (\{X\} \rightarrow \{f(X)\})\ \textit{in}\ \{S\}$$

The same iteration has the more interesting form:

$$\textit{rewrite}\ (X \rightarrow f(X))\ \text{in}\ S$$

which requires to apply non-deterministically the rewriting $X \rightarrow f(X)$ to any part of S until stabilization; and, thanks to the monotonicity of f, the final value will be the same as in the previous *rewrite* rule, independently of the order in which rewritings are applied. To this non-deterministic algorithm corresponds a family of deterministic algorithms, one of them being the naive classic solution (the considered subpart is the whole set each time), the other ones explicity using purely local transformations.

A kind of analogy can be found between the *rewrite* operator and some mechanisms in Artificial Intelligence systems which continuously re-interpret their data. As a matter of fact we have been influenced by the efforts towards extending term-rewriting to new objects, especially graphs, and also by the Γ combinator of Banatre and LeMetayer [BaL, 86] presented within the framework of an FP-like language working with multisets instead of sets. The use of multisets introduces important differences. For instance a modification of a part of a multiset implies a modification of the whole multiset; of course that is not the case for a set: for instance, substituting $f(P)$ to P in S has no effect upon S if $P \subseteq f(P) \subseteq S$. Some problem descriptions have a better fit with a multiset representation and some other ones with set representation.

Let us consider a few examples:

(8) $$\textit{rewrite}\ (\{a\} \rightarrow \{\})\ \text{in}\ S$$

suppresses a from S

(9) $$\textit{rewrite}\ (\{\} \rightarrow \{a\})\ \text{in}\ S$$

adds a to S(5)

(10) $$\textit{rewrite}\ (\{X, Y\} | X < Y \rightarrow \{Y\})\ \text{in}\ S$$

(5) In that case, the *rewriting* is applied only once, since the change condition may be satisfied at most once.

given an ordering $<$ on S, this will construct a set containing as elements the $\max_< S$. This algorithm is the least deterministic possible algorithm: it suppresses from S any element for which a greater element is known to exist. Eventually, only the $\max_< S$ elements are left.

The next example, inspired by an example in [BaL, 86], is the topological sort of a set S, according to a given map or binary relation R. We use here pairs or 2-uples with the usual set theoretic meaning: $[a, b]$ stands for $\{a, \{a, b\}\}$. R is described as a set of pairs, e.g.: $\{[a, b], [c, a], \ldots\}$ means that rank of b must be greater than rank of a, rank of a than rank of c etc... First of all let us consider an arbitrary enumeration function on sets *enum* (S) which returns any given set S as an enumeration, i.e. as a tuple, in any order. For instance: enum $(S) = [x : x \in S]$. The iterator semantics for $x \in S$ is that of the $\forall$ operator on sets, i.e. one of arbitrary selection at each iteration step. Instead of the tuple notation for unlimited lists we use the map notation:

$$t := [a, b, \ldots, g] \rightarrow t' := \{[1, a], [2, b], \ldots [\#t, g]\}$$

which associates to each tuple element a pair made of the rank of the element and the element itself. Let S' be an enumeration of S under this form. The topological sort algorithm reads:

$$\text{(assuming } [X, Y] \in R, [i, X] \in S', [j, Y] \in S')$$

$$\textit{rewrite}\,(\{[X, Y], [i, X], [j, Y] | i > j\} \rightarrow \{[X, Y], [j, X], [i, Y]\}) \text{ in } R \cup S' \tag{11}$$

The transitive closure algorithm (4) may then be written:

$$(\textit{rewrite } (Y \text{ with } \{Y, D\} \rightarrow Y \cup D) \text{ in } X \cup f) - f \tag{12}$$

where f is considered a map, i.e. a set of pairs. In others words, in presence of its origin (the set Y), a functional dependency – more precisely the corrsponding singleton set – is rewritten into its target (the set D). From this specification one immediately sees that a functional dependency is only used once! This observation, important for efficiency, is not easily inferred from the expression in (4). It is relatively easy to show that both solutions identify the same set of attributes.

The *rewrite* operator aims at the definition of fixed-points of ordinary set functions – on finite sets – through local transformations. The *rewrite* operator allows the description of transitions among sets with a very high level formalism – still an operational description. As we have seen, a direct translation in SETL of such specifications is possible. In this context, it is naturally much more difficult to synthetize a rewriting algorithm from a specification that it is in the doubly monotone context of RAPTS. However the ensuing optimizations – e.g. finite-differencing, data structure selection – remain the same. These optimizations have been developed until now by Paige's team to transform SQ+ specifications into efficient imperative programs executing on traditional – i.e non-parallel – machines. We are studying their adaptation to the *rewrite* context. We are also studying under what conditions it is possible to transform non-monotone *rewrite* operations into monotone fixed-point iterations.

6. Optimizing set computations-1: finite differencing.

Let us expose briefly the principles behing the finite-differencing optimization, one of the major techniques used in RAPTS [Pai, 84] [Pai, 87]. Let EXP be a set expression of high evaluation cost, repetitively evaluated in a given loop. Let E be an auxiliary variable and consider maintaining the invariant $E = EXP$ whenever EXP is evaluated. It will then be possible to replace the evaluation of EXP by an access to the value stored in E. To maintain the invariant $E = EXP$ one has to:

– compute EXP and assign its value to E upon entering the loop

– propagate to E any modification of a parameter of EXP during the loop execution. This propagation takes the form of difference code for EXP wrt. the parameter modification. An example will illustrate the process. Let us consider a program fragment (on the lhs) and its transform (on the rhs) which exhibits 2 invariants associated to the auxiliary variables D and E:

(13)

```
                                      $D := \#\{x \in A|K(x)\}$;
                                      $E := \{x \in S|C(x)$ and $x \notin A\}$;
(while $\exists x \in S|C(x)$ and $x \notin A$)   (while $\exists x \in E$)
                                         $E$ less := $x$;
   $A$ with := $x$;                         $A$ with := $x$;
                                         if $K(x)$ then $D$+ := 1; end;
   print($\#\{x \in A|K(x)\}$);            print($D$);
end;                                  end;
```

In italic we have displayed the code needed to maintain the two invariants. This transformation is an optimization only if the cumulated cost of executing the difference code is less than the cost of re-evaluating the expressions. In that case the expressions are considered *differentiable* wrt. the modifications. To that effect an evaluation of the asymptotic computing cost is performed, by induction on the structure of the expressions. This technique is applied to all the sub-expressions of a given complex expression: thus a cascade of maintenance code and invariants will be introduced. It should be stressed that in RAPTS the system transforms into a normal form all the considered expressions in order to exhibit a maximum number of differentiable subexpressions. In the attribute closure example (5) the sub-expression $Y \subseteq S$ of $f[\{Y \in$ domian $f|Y \subseteq S\}] - S$ will be rewritten: $\#\{z \in Y|z \notin S\} = 0$. Thus the invariant $\#\{z \in Y|z \notin S\}$ will be maintained for each Y in domain f. Whence the propagation of the extensions of S to the triggering of functional dependencies will be assured.

This technique could be used outside the program synthesis context: the user may declare his intention of maintaining a variable equal to a given expression, within a given program region, and supplies the appropriate difference codes. The system will then insert the appropriate code in all the needed places [Pai, 86]. This is a fairly general and particularly powerful technique which was applied in a restricted form (strength reduction) in some compilers. Applications to the maintenance of views in databases are also possible [Pai, 84].

7. Optimizing set computations 2: data structure choice.

There are many possible representations for sets. This may be perceived as a disadvantage, since no a priori computer representation exists, unlike for lists. It has its advantages: it provides a great freedom of implementation which could yield in the end very different algorithms. A common default representation consists in a doubly linked list combined with a hash-table. The goals for an efficient representation are to minimize:

1 the systematic traversal of sets,

2 the cost of associative accesses (e.g. membership tests, access to the domain of a map for finding the range element $f(a)$ of a given map f)

3 data redundancy: storing only once values which are shared by several sets(6)

The default representation satisfies goal 1, more or less goal 2 and absolutely not goal 3.

In order to get a good understanding of the data structure choice mechanism, a good clasisfication of associative accesses is needed. The following example illustrates all the access patterns we are interested in:

(6) the languages semantics is a *copy-value* semantics which entails practice may duplications of data values, e.g for each assignment $t := z$;

syntactically detectable access patterns:

(14)

```
$\forall x \in S \ldots$                  $x$ retrieved from $S$
  $Q$ less := $x$;                $x$ used to access $Q$
  $S$ less := $x$;                $x$ used to re-access $S$
  if $x \in R$ and $x \notin T$    $x$ used to access $R, T$
    then $T$ with:= $x$;          $x$ added to $T$
  end if;
end $\forall$
```

In (14) S could be traversed like a list, and access to its elements is sequential (7). However access to Q, R, T should be at constant cost. This could be possible if the value x were stored at a location from which the value of the expressions: $x \in Q$, $x \in R$, $x \in T$ could be directly computed without using x itself. The following organization of the data may satisfy these requirements:

(15)

values of elements of B	Q	R	T
a	1	0	0
b	0	1	0
c	0	1	1
...	...	...	...

B, in the table (15), is a set which could be seen as the union $S \cup Q \cup R \cup T$. The 1_s and 0_s on each row indicate whether the corresponding value is a member of the set indicated in the column. For instance the 1st row of (15) reads: $a \in Q$, $a \notin R$, $a \notin T$. S could be a linked list of pointers to the appropriate elements of B. B is called a base. Sets implemented as Q, R, T are called strongly based whereas S is known as weakly based.

A base is a union of sets exchanging data. Dynamic bases are bases whose set of stored values changes at execution-time. Dynamic bases have been found to have poor performances. Therefore RAPTS considers only static bases, i.e. made of elements all present in the input data of the program. Individual sets are modifiable, provided

(7) Ideally, as suggested in [DaM, 86], access to all the elements of S could be in parallel.

they take their elements from a fixed set of values determined at the end of the data input phase. Sets which could not be based will be implemented with the default implementation.

From an analysis of the set operations on a given (SETL, *SQ*+) program it is possible to determine the bases and the type of representation (weakly-, strongly-based) for the basic sets. Such a technique is partly integrated into the SETL optimizer [Fre, 83] which allow the user to define the bases for specific sets. A fully automated version associated with *SQ*+ has been designed. That version has been extended and reformulated as a type inference problem by one of the authors [Fac, 88], [CFHPS, 88] who implemented it in PROLOG. This is what will be discussed in the following section. We discuss first how we have used these techniques in the SED project.

The SED project is an ESPRIT project for experimenting software engineering, and specifically prototyping with SETL. A basic language environment for SETL [DDFJ, 87] has been build with MENTOR [DoKLMM, 83]: the SETL programs are represented as abstract syntax trees, annoted with type information; this type information is computed by a type inference system build with the help of TYPOL [Des, 86] [Kah, 87]. TYPOL is a language for specifying a program semantics as a syntax-directed pattern matching inference system. The pattern matching capability of TYPOL mixes MENTOR pattern matching in trees with PROLOG unification. Around this MENTOR-TYPOL kernel a number of tools have been constructed. Among these, an adaptation of RAPTS and a fully operational, yet non-optimized SETL-to-Ada traslation system [Dob G, 87]. A natural objective consists in optimizing the Ada generated code. The data structure scheme we have exposed has proved quite effective on SETL programs as well as on Ada programs representing the translation of SETL programs. A hand simulation of the proposed data-structure selection method combined with effective performance measurements [CFHPS, 88] has confirmed the importance of such an optimization in the SETL to Ada translator.

It will be shown later that data structure choice as exposed in 7 could be seen as an extension of type inference to an extended type model, provided a special kind of sub-typing discipline is exercised. In the SED case, the type inference system for SETL described in [DDFJ, 87] was used as the starting point. Unlike the type and data structure determination system for SQ+, the type determination for full SETL programs may only provide approximate results, because the weak typing. This is why it is expected that data structure choice considered as an extension of the type-finder for SETL would be less refined than for SQ+. At the present time this facility is under development in the SED environment.

8. Inferencing representations for sets.

The algorithms developed until now (e.g. [Fre, 83]) for determining the bases and the associated set representations use a value-flow analysis, quite different from – but still in the same spirit as – classical data flow analysis: these algorithms compute all the relations existing among variables, e.g.: adding X to S, testing for the membership of X in S, etc. Starting from these relations, these algorithms compute the bases in a succession of merge operations: informally, as soon as two sets communicate, they merge their bases.

We have formalized this problem [Fac, 88] [CFHPS, 88] in Natural Semantics, a formalism originating with the work of Plotkin [Plo, 81] which was developed at INRIA with the TYPOL language. Its root principle is the construction of a formal system in the style of Gentzen's natural deduction. This system axiomatizes the relations which should hold between syntactic objects, here SQ+ (or SETL) abstract syntax trees and the semantic values, here the properties of bases. The system must generate sequents of the form $E| - P : V$ meaning: under the hypothesis E, or, in the environment E, P has semantic value V. These sequents are generated by inference rules, constituting a systematic induction on the abstract syntax. E.g. a simple type inference rule for SQ+ reads:

$$\frac{E| - X{:}T \; E| - S : \mathrm{set}(T)}{E| - X \in S : \mathrm{bool}} \tag{16}$$

Rule (16) formalizes the following: if, with the hypothesis E on types, it is possible to prove that the expression X is of type T, and that S is of type set (T), then, under the same hypothesis it is possible to conclude that the boolean operation $X \in S$ is well typed with type boolean.

Conversely the proof that a given SQ+ expression is well typed in an unknown «environment» E, will intanciate E to a set of assertions on the variables types. A type inference for SQ+ and another for SETL have been been designed in this way.

A very fruitfull idea consists in reducing the computation of bases to a type inference problem. We consider bases expressions comparable to type expressions, using variables for unknown bases, and we build a formal system allowing a constructive proof that a given program is well-based. This proof will supply, in the end, the bases and the representation as an association between identifiers and base expressions. Among such associations there will be terms like:

S : set(B)	S is a set based upon B
X : B	X is an element of the base B

An essential property is the

subtyping rule #1:

(17a) if both, a base and a type expressions can be proved for the same X:

$X : B$, $X : t$ (e.g X: integer)

then, under the strong typing hypothesis([8]), any element of the base B has type t (e.g. integer), i.e. the sub-type relation

$B \leq t$ (e.g. $B \leq$ integer)

([8]) The strong typing hypothesis implies the homogeneous charcter of a base: all the elements of a given base have the same representation.

holds.

A contrario, if one proves that $X : t$ (e.g. X: integer) holds but no relation of the kind $X : B$, the X will be of type t (e.g. integer) but not a member of a base. This will be the case if X is a value created from the evaluation of an expression (see below the example (18) with an addition expression) since we have excluded dynamic bases.

Similarly $S : B1$ S :set($B2$) means that S is an element of the base $B1$ and also that S is based upon $B2$. Thus all the elements of $B1$ are based upon $B2$. This is again a subtype property: $B1 \leq$set($B2$). Usually the notion of subtype is used to model type conversions and coercion rules or inheritance properties. The only aspect which is relevant to our subtyping is the:

subtyping rule #2:

(17b)

$$\frac{ENV| - EXP : t1 \; ENV| - EXP : t1 \leq t2}{ENV| - EXP : t2}$$

meaning that if an expression must be of a given base or type $t2$, in the context associated to the set of constraints («environment») ENV, then every sub-type $t1$ of $t2$ is also acceptable.

This rule is actually present in our system, and states that it is possible to abandon a base and still get a correct (less refined) basing. In fact the «finest» type model corresponds to a partition of sets into the smallest possible bases, the «coarsest» model to a simple type inference without bases. By adding rule (17) to a classical type inference system for a strongly typed version of SQ+, one expresses base inference (without the selection between strongly and weakly based) as the «finest» type model in the new system. For instance, rule (16) remains valid if T denotes a base. It has as new interpretation, that X is an element of the base upon which S is based. However let us consider the typing rule:

(18)
$$\frac{ENV| - X : \text{integer} \; ENV| - Y : \text{integer}}{ENV| - X + Y : \text{integer}}$$

If X and Y are elements of an integer base B then X: integer, Y: integer will result from $X : B$, $Y : B$, $B \leq$integer, according to

(17b). However, for $X + Y$ no rule will guarantee that $X + Y : B$. If for instance this expression is assigned to a variable Z, then the assignment rule will force the type of Z to integer: it's value being computed Z cannot be a base element. Furthermore, the rule for expressing the constraints on inserting an element into a set will entail that a set containing Z as element cannot be based.

It is interesting to note that this kind of interaction information is implicitely computed by a simple type inference system but is rejected in the final type determination because of constants in type expressions. Here is an example: given sets S and T, let us assume that one has determined that S is of type set (integer) and that because of interactions between S and T, T has the same type as S, then type inference will infer:

(19) $$\text{type}(S) = \text{type}(T) = \text{set(integer)}.$$

However, if the type of S and that of T have been established independently of one another to be, in both case, set(integer), then type inference will infer (19) too. But in the first case (19) means:

$$S : \text{set}(B), T : \text{set}(B),\ B \leq \text{integer}$$

whereas in the second case:

$$S : \text{set}(B1), T : \text{set}(B2),\ B1 \leq \text{integer}\ B2 \leq \text{integer}$$

The rule to keep the information about type is a reformulation of sub-typing rule #1 (17a): to infer $S : t$ it suffices to infer $S : B$ for some $B \leq t$. An equality between subtypes will be generated only if the corresponding sets are interacting.

This notion of sub-types has allowed us to use the same framework for computing several relations among bases which were not accounted for in the usual base determination algorithms.

We have implemented this system in PROLOG, working directly on the SQ+ abstract syntax terms. The main difficulty was to obtain a deterministic system computing the bases directly from the formal inference system – for which these bases were just one of the possible solutions. We succeeded essentially by adding two rules derived from the system and by ordering the clauses correctly. The resulting PROLOG program then generates as first solution the finest possible model, then solutions with fewer and fewer bases, until it produces the coarsest solution, corresponding to «pure» type inferencing and no bases. Base merging corresponds in this program to PROLOG unification. The rules are somewhat more complex for the exact representations (i.e. strongly- or weakly- based sets), but they have been designed in the same style by refining the preceding rules.

A version of this base inferencing system implemented in SETL, using specific pattern matching in trees instead of PROLOG, will be incorporated to RAPTS.

As explained in 7 a TYPOL version of this sytem, but for full SETL instead of $SQ+$ is being implemented, using as starting point the type-finder for SETL developed for the SED project. Subtyping rule #2 garantees correctness even with our approximate handling of type-finding. This will eventually support a corresponding optimization of the SETL-to-Ada translation.

9. Conclusion.

The set oriented notations present essential qualities for programming. They correspond to a widespread formalism with rigorous and well known foundations. If such notations become executable they become a formidable tool for prototyping. Their importance increases with the new data models, able to take into account set oriented objects.

To achieve a real programming tool, with acceptable performance, without limiting the spectrum of set oriented constructs and operations, deep transformations and optimizations are required.

We have presented some of those. They aim at the production of efficient imperative code from very high level specifications. Other approaches are possible, and some of the exposed techniques could probably be applied in other contexts. The initial results are extremely encouraging, and automatically generated code from specification has been demonstrated to be as efficient in some complex cases as carefully hand crafted programs.

It is probable that the development of this kind of work will stimulate the use of set oriented operators in programming languages.

10. acknowledgements.

This work owers enormously to the many discussions we had with Bod Paige, Fritz Henglein, and Jaizhen Cai. And also with Veronique Donzeau-Gouge, Catherine Dubois and Thierry Despeyroux. Françoise Jean, Jacques Leger, Ulrich Gutenbeil and Wolfgang Franke have actively participated to the implementation of the presented transformation system and of its ensuing experimentation.

REFERENCES

[1] [Abi,87] Abitboul S., Grumbach S., *Bases de donneés et objets structurés,* TSI 6, 5, (1987), 383-404.

[2] [AhU,79] Aho A.V., Ullman J.D., *Universality of data retrieval languages,* Proc. 6-th POPL, 110-120.

[3] [BaL,86] Banatre J.P., LeMetayer D., *A new computational model and its discipline of programming,* Res. Rep. 566, INRIA, sept. 86.

[4] [BalG,88] Balzer R., Gabriel R.P., (editors) *Draft Report on Requirements for a Common Prototyping,* System, Darpa-ISTO, Nov. 88.

[5] [Cai,87] Cai J., *An Iterative version of Hopcroft and Tarjan's Planarity Testing Algorithm,* Tech. Rep. 324, NYU, Courant Institute, 1987.

[6] [CaiP,88] Cai J., Paige R., *Program Derivation by Fixed Point Computation,* to appear in Science of Computer Programming, 1989. (avail. as IBM report RC 13947, 23 Aug. 88).

[7] [CFHPS,88] Cai J., Facon Ph., Henglein F., Paige R., Schonberg E., *Type transformation and Data Structure Choice,* submitted., NYU-CS, Nov. 88.

[8] [DaM,86] Dahlhaus E., Makowsky J.A., *The Choice of Programming*

Primitives for SETL-like languages, ESOP 86, LNCS #213, Springer-Verlag.

[9] [DDFJ,87] Donzeau-Gouge V., Dubois C., Facon Ph., Jean F., *Development of a Programming Environment for SETL,* Proc. ESEC 87, also in LNCS #289 21-32.

[10] [Des,86] Despeyroux Th., *TYPOL a formalism to implement Natural Semantics,* RR INRIA, Sophia-Antipolis, May 86.

[11] [DobG,87] Doberkat E.E., Gutenbeil U., *SETL to Ada tree-transformations applied,* Information & Software Technology, 29, 10, dec. 87, 21-32.

[12] [DoKLMM,83] Donzeau-Gouge V., Kahn G., Lang B., Melese B., Morcos E., *Outline of a Tool for Document Manipulation,* IFIP 83, North-Holland Elsevier 615-620.

[13] [Fac,88] Facon Ph., *Langages ensemblistes et transformations de programmes,* IIE report, nov. 88.

[14] [Fac,88-2] Facon Ph., *Langages ensemblistes et transformations de programmes,* 10th STFI, Tunis, mai 89.

[15] [Fre,83] Freudenberger S., Schwartz J.T., Sharir M., *Experience with the SETL Optimizer,* ACM TOPLAS 5, 1, (1983) 26-45.

[16] [Kah,87] Kahn G., *Natural Semantics,* RR INRIA #601, feb. 87.

[17] [Lac,88] Lacrampe J.J., *S3L: un langage sur les collections pour la manipulation fonctionnelle des ensembles,* Journées GROPLAN: les languages et leurs environnements, 1988.

[18] [Naf,86] Naftalin M., *An experient in practical Semantics,* ESOP 86, LNCS #213, Springer-Verlag.

[19] [Pai,84] Paige R., *Applications of finite-differencing to Database Integrity control and query/transaction optimization,* in Advances in Database Theory, vol. 2, Gallaire-Minker-Nicolas editors, Plenum Press, 1984.

[20] [Pai,86] Paige R., *Programming with Invariants,* IEEE Software, 3, 1, jan. 86, 56-89.

[21] [Pai,87] Paige R., Henglein F., *Mechanical Translation of Set Theoretic Problem Specification into Efficient RAM Code- A Case Study,* J. Symbolic Computation 4, (1987) 207-232.

[22] [Plo,81] Plotkin G., *A Structural Approach to Operational Semantics,* DAIMI Fn-19 Notes, Aarhus University, Danemark.

[23] [SchD,86] Schwartz J.T., Dewar R.B.K., Dubinsky E., Schonberg E., *Programming with Sets, an Introduction to SETL,* Texts & Monog. in Comp. Sci., Springer-Verlag, 1986.

[24] [Smos,88] Smosna M., *Design of a new Ada front End for SETL,* private comm., 1988.

Philippe Facon CNAM-IIE
18 allée Jean-Rostand BP 77
91002 Evry Cedex-France

Yo Keller KEPLER
8 rue des Haies
75020 Paris-France