A Language for Semantic Analysis

Jiazhen Cai

Technical Report 635

May 1993

New York University
Department of Computer Science
Courant Institute of Mathematical Science
251 Mercer Street, New York, NY 10012

A Language for Semantic Analysis

Jiazhen Cai [*]

New York University/Courant Institute
New York, NY 10012
and
Purdue University
West Lafayette, IN 47907

ABSTRACT

Semantic analysis is important for compilers. In the APTS program transformation system, semantics is specified by rules in the language RSL. The semantic rules are interpreted by APTS to generate the semantic information of the program, which is then used by the rewriting engine for program translation. This approach has proved to be convenient and powerful in our construction of a SETL-to-C compiler. In this paper, we discuss the features, applications, implementation strategy, and performance of RSL.

1. Introduction

RSL is the specification language of the APTS system, an experimental program transformation system on which Robert Paige and the author have been working for several years. Recently, a SETL-to-C translator, written in RSL, has been built in the APTS environment with some success [4]. SETL is a very-high-level language and is convenient to use, but usually much slower than C. With the SETL-to-C translator, we combine the convenience of SETL with the efficiency of C.

Fig. 1 illustrates how the SETL-to-C translator (also called the SETL Accelerator) is built on the top of APTS. Considering the importance of semantic information to the high level language translation and optimization, we designed an inference engine in APTS to perform the semantic analysis. Semantics are specified by RSL rules. By applying the semantic rules on the parse tree of the input program, the inference engine computes the semantic information and stores it as database relations, which are then used by the rewriting engine to transform the input program. The actions of different modules are coordinated by a control file through the command

* The research of this author was partially supported by National Science Foundation Grant No. CCR-9002428, MIP-9300210, and by Air Force Office of Scientific Research Grant No. AFOSR-91-0308.

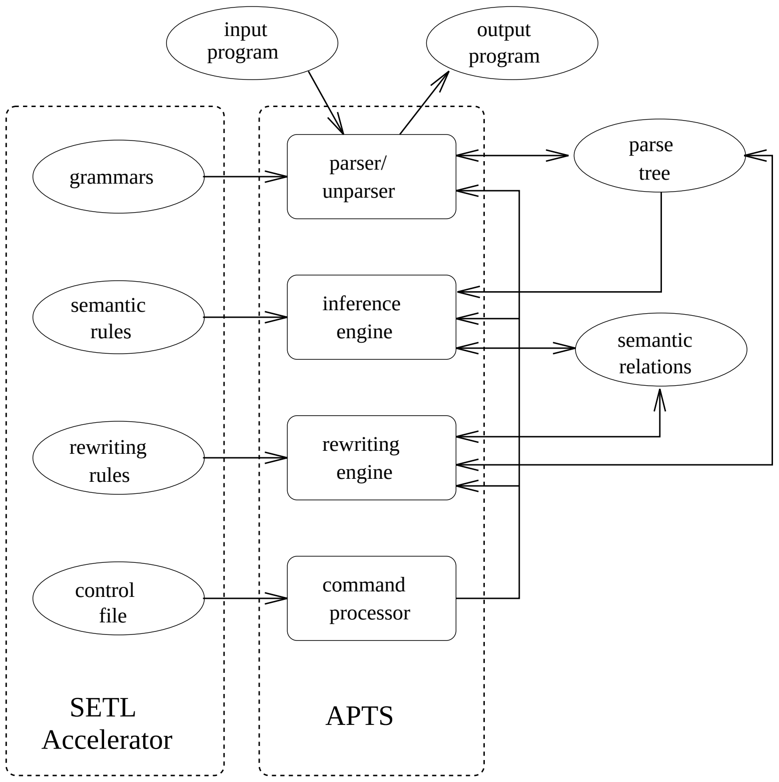

Fig. 1. Configuration of APTS and SETL Accelerator

processor.

RSL was first designed for specifying semantic rules, and then extended to become the specification language of the APTS. In this paper, we will only discuss the subset of RSL which is used for semantic specifications. We argue that RSL is more convenient and powerful than the attribute grammar in specifying semantics.

2. The language

2.1. Rules and Transcripts

In APTS, semantics are specified in rules, and rules are organized into *transcripts*, as shown in Example 1.

A transcript consists three parts: a header, a declaration list, and a transcript body. In this example, the header is the first line, giving the name of the transcript *sg*; the declaration list contains one *rel* declaration, which declares three relations: a unary relation *person* of strings and two binary relations *parent* and *same_generation* of pairs of strings; and the body contains four rules, each having the form *left-hand-side* → *right-hand-side*. The first rule says that $p1$, $p2$, $p3$, $p4$ and $p5$ are all persons. The second rule says that $p1$ is the parent of $p2$, and so on. The last two

Example 1. A transcript

```
transcript sg;
rel person: [string];
    parent, same_generation: [string, string];
begin
     true
     ->  person(p1) and
         person(p2) and
         person(p3) and
         person(p4) and
         person(p5);

     true
     ->  parent(p1, p2) and
         parent(p1, p3) and
         parent(p2, p4) and
         parent(p3, p5);

     person(.x)
     ->  same_generation(.x, .x);

     parent(.x, .y) and
     parent(.z, .w) and
     same_generation(.x, .z)
     ->  same_generation(.y, .w);
end;
```

rules defines the relation *same_generation*: two persons are of the same generation if they are the same person (rule 3) or their parents are of the same generation (rule 4). In RSL, variables are prefixed by ".".

The syntax and semantics of the rules are more or less conventional. Each left-hand-side is either a boolean constant **true**, a *left-term*, or several left-terms connected by logical connectors **and**, **or**, or **not**.

Each left-term has the form $R(p1,...,pk)$, where R is either a system defined predicate or a user declared relation, $p1,...,pk$ are parameters, and k is the arity of R. A *left-term* can be evaluated only when each of the free variables contained in the parameter list is bound to some constant. In case R is a user declared relation, the left-term $R(p1, ..., pk)$ is true if and only if the tuple $[p1, ..., pk]$ (after variable substitution) is in R.

Each right-hand-side can be one *right-term* or several right-terms connected by **and**. Each right-term specifies an action and also has the form $R(p1, ..., pk)$, where R can be either a system defined procedure or a user defined relation name. In case *R* is a user defined relation, the action *R*(*p*1, ..., *pk*) will insert the tuple [*p*1, ..., *pk*] into *R*, if it is not there already.

A term *R*(*p*1, ..., *pk*) is called a *relational term* if *R* is a user declared relation name. Notice that a relational term in the left-hand-side of a rule is interpreted differently than it is in the right-hand-side.

During the execution of a transcript, a rule becomes *active* if its left-hand-side evaluates to **true** with respect to some variable binding. A variable binding that makes the rule's left-hand-side *true* is called an *activating binding* of this rule. If *r* is an active rule, and *B* is its activating binding, then we call the pair (r, B) an *active instance* of *r*. In general, there may be more than one active instance at a time. The inference engine will choose one active instance (*r*, *B*) at a time nondeterministicly and perform the right-hand-side actions of *r* with respect to *B*. An active instance may become non-active and vice versa because of the actions. Each active instance will be selected by the inference engine only once before it becomes non-active. The execution of a transcript terminates when all the right-hand-side actions of the active instances are performed, and none of them can change the current values of user defined relations.

When the transcript *sg* in Example 1 is executed, the resulting relation *same_generation* will be { [p1, p1], [p2, p2], [p3, p3], [p4, p4], [p5, p5], [p2, p3], [p4, p5], [p3, p2], [p5, p4]}.

2.2. External relations

In a transcript *T*, relations declared in the *rel* declaration are called the *local relation*s of *T*, or the relations *defined* in *T*. One transcript can make reference to relations defined in other transcripts by declaring them to be *external.* For example, the transcript in Example 1 can be broken into three transcripts using the *external* declaration, as shown in Fig. 2.

Relations used in a transcript must be declared as either local or external. A transcript can only modify its local relations.

Each transcript can be parsed separately. Parsed transcripts can be unloaded to a file and loaded back to the system later. When a transcript is executed, the system will first compute its external relations recursively before all the local relations are computed.

Let R_1 and R_2 be two relations declared in a transcript *T*, and let *r* be a rule in *T*. If R_1(...) is a left-term of *r*, and R_2(...) is a right-term of *r*, then we say R_2 *depends* on R_1. We define the dependency graph of a set of transcripts *V* to be $G = (V, E)$, where $E = \{ [x, y]: x, y \in V$, an external relation of transcript x is defined in the transcript $y\}$. The current implementation requires that *G* be acyclic. This means that if two relations recursively depend on each other, then they must be defined in the same transcript.

```
transcript sg;
rel same_generation: [string, string];
external person: [string];
         parent: [string, string];
begin
    person(.x)
    -> same_generation(.x, .x);

    parent(.x, .y) and
    parent(.z, .w) and
    same_generation(.x, .z)
    -> same_generation(.y, .w);
end;
```

```
transcript person;
rel person: [string];
begin
    true
    -> person(p1) and
       person(p2) and
       person(p3) and
       person(p4) and
       person(p5);
end;
```

```
transcript parent;

rel parent: [string, string];

begin
    true
    -> parent(p1, p2) and
       parent(p1, p3) and
       parent(p2, p4) and
       parent(p3, p5);
end;
```

Fig. 2. External relations

2.3. Tree patterns

To specify semantic properties of a language, we need some way to make reference to syntactic objects. In principle, a parse tree is just a set of relations, and so syntactic objects can be referred to using relational terms. But this approach is both inconvenient to use and inefficient to implement. In RSL, the syntactic objects are referred to as tree patterns and accessed through the built-in predicate *match*. For example, we can use *match*(%*expr*,.*x*+.*y*%) to find a node in the parse tree that represents an expression with an operator "+". We use a pair of "%" to delimit the tree pattern and use a "|" to connect the *match*-term with the rest of the left-hand-side.

Example 2 is a small portion of the transcript *type* for type analysis, which shows the usage of tree patterns and the *match* predicate. In this example, the relation *type* is declared to be a binary relation between *tree*s and *string*s. A *tree* is just a subtree of the parse tree.

The first rule says that if .*x* is any expression in the parse tree, and its lexical type is *int*, then the type of .*x* is *int*. Here, *eq* is a system predicate that tests the equality of its two

Example 2: Tree patterns and the *match* predicate

```
transcript type;
rel type: [tree, string];
begin
    match(%expr, .x%)
    |  eq(lextyp(.x), int)
    -> type(.x, int);

    match(%expr, .x + .y%)
    |  type(.x, .t) or type(.y, .t)
    -> type(%expr, .x + .y%, .t);

    match(%expr, .x + .y%)
    |  type(%expr, .x + .y%, .t)
    -> type(.x, .t) and type(.y, .t);

-- inference rule for other cyntectic constructs
-- ...
end;
```

arguments, and *lextyp* is a system function that returns the lexical type of its argument. The information about lexical type is computed by the lexical scanner and stored with the parse tree. The other rules are self-explanatory.

The lines introduced by the "--" signs are comments and are ignored by the system.

When applied to the parse tree $x+y+1$, the transcript will yield the following information: *type* = {[1, int], [x+y, int], [x, int], [y, int], [x+y+1, int]}

2.4. Type declaration

We require that the types of the parameters of each user defined relations be declared. Right now, the system only supports a limited number of types. Some most frequently used types are: *string*, *list*, *tree* and *node*. We have seen the types *string* and *tree* in the previous examples. The type *node* is also used to represent subtrees in the parse tree. A *node* represents an occurrence, but a *tree* represents a common subtree. In other words, different occurrences of the same subtree are represented as different *node*s, but the same *tree*. Internally, *tree*s are represented as value numbers, but *node*s are represented as pointers to the root of the subtree.

A *list* can be either an empty list [], a string, or a list of one or more *list*s. *List* can be used to encode structured information. For example, we can use [*set*, *t*] to encode the type of sets whose members have the type *t*, where *t* is a string encoding a type variable, and use [*set*, [*tuple*, [*t*1, *t*2]]] to encode the type of sets of tuples whose first component is of type *t*1 and second component *t*2. A *string* is a special case of a *list*. Thus, it would be more convenient to declare the type of the relation *type* to be [*tree*, *list*] in Example 2.

2.5. Type conversion

The system supports limited run-time conversion between types. Consider the following example:

Example 3. Type conversion

```
transcript type_convert;
rel node_rel: [node];
    tree_rel: [tree];
    string_rel: [string];
    list_rel: [list];
begin
    match(%expr, .x+.y%) -> tree_rel(%expr, .x+.y%);
    tree_rel(.x) -> node_rel(.x);
    tree_rel(.x) -> string_rel(.x);
    string_rel(.x) -> tree_rel(.x);  -- run time error
    string_rel(.x) -> list_rel([.x, [.x]]);
end;
```

In the first rule, the pattern *.x+.y* is first instantiated with a *node* and then converted into a *tree* when stored into *tree_rel*. In the second rule, the conversion goes the other way around. In general, *tree*-to-*node* conversion is not unique and should be used with caution. In the third rule, a *tree* is converted to a *string* by unparsing. But *string* to *tree* conversion, as used in the fourth rule, is not allowed. There is no conversion in the last rule.

2.6. Incremental operations

Consider the transcript in Example 4. When executed, it will give the following results:

addition_rel: { [*a*, [*b*, _*v*1]], [*a*, [_*v*2, *c*]] }

replace_rel: { [*a*, [*b*, _*v*1]] } or { [*a*, [_*v*2, *c*]] }

unification_rel: { [*a*, [*b*, *c*]] }

The two tuples in *addition_rel* are added by the first rule. By default, the incremental operation of the relation *addition_rel* is *insert*, *i. e.*, when a new tuple is added to *addition_rel*, the old tuples in it will remain unchanged. Although the same two tuples are also added to *replace_rel*, only one remains, since the incremental operation of *replace_rel* is declared to be *replace*, and the first

Example 4. Incremental operations

```
transcript incremental_ops;
rel addition_rel, replace_rel, unification_rel:
        [string, list];
key replace_rel, unification_rel: [1];
incremental replace_rel: replace;
            unification_rel: unify;
begin
    true
    -> addition_rel(a, [b, _v1]) and
       addition_rel(a, [_v2, c]);

    addition_rel(.x, .y)
    -> replace_rel(.x, .y) and
       unification_rel(.x, .y);
end;
```

components of its tuples are declared to be *key*s. Thus, when a new tuple is added to *replace_rel*, the system will remove from *replace-rel* the old tuple with the same first component. Similarly, since the incremental operation of *unification_rel* is declared to be *unify*, and the first components of its tuples are declared to be the *key*s, the system will unify the the new tuple with the old one in *unification_rel* that has the same first component as the new one. The identifiers with a prefix "_" are considered unification variables.

3. Questions and answers

Following are the RSL solutions to some of the frequently asked questions about rule-based systems.

3.1. Safe Rules

The set of activating bindings of a rule is called the *conflict set* of the rule. A conflict set can be infinite, as in the rule

not *person* (.*x*) -> *non_person* (.*x*);

We call such rules *unsafe*. A rule is *safe* if its conflict set is always finite. We want to avoid dealing with unsafe rules and give a simple sufficient condition for safe rules.

A left-term is *positive* if it is in the scope of even number of negations, and *negative* otherwise. A variable occurrence in some parameter of the *match* predicate or a positive relational left-term is called *binding occurrence* if it is not also contained in any function applications.

Let r be a rule. Let *binding_var*(r) be the set of variables that have binding occurrences in r. Let *all_var*(r) be the set of all variables in r. Obviously, if *binding_var*(r) = *all_var*(r), then r is safe. Rules satisfying such condition are *acceptable.* Our system only accepts *acceptable* rules. For example, the following rule is acceptable

living_being$(.x)$ **and not** *person*$(.x) \rightarrow$ *non_person*$(.x)$;

where *living_being* is a user defined relation.

3.2. Termination

It is easy to write a transcript that does not stop even if all the rules are acceptable:

```
transcript non_stop;
rel list_rel: [list];
begin
     true -> list_rel(1);
     list_rel(.x) -> list_rel([.x]);
end;
```

which defines the infinite unary relation { [1], [[1]], [[[1]]], ...}

However, **if all the rules are acceptable, and the incremental operations** *replace* **is not used, and if all the parameters to the relational right-terms are either constants or variables, then the termination is guaranteed.**

The system is not responsible for the termination.

3.3. Fixed point

Even if a transcript terminates, its result may not be unique:

```
transcript non_unique;
rel R1, R2: [string];
begin
     not R1(1) -> R2(1);
     not R2(1) -> R1(1);
end;
```

If the first rule is fired first, the result would be r1 = {}, r2 = {[1]}; otherwise, the result would be r1 = { [1] }, r2 = {}.

Consider the execution of a transcript T, which defines the local relations $R1, ..., Rk$. Let $R = \{ [i, x]: i = 1...k, x \text{ in } Ri\}$. Let $next(R)$ be the new value of R resulting from a right-hand-side action of some active rule. Then $next(R)$ is a nondeterministic function of R. Let $NEXT(R)$ be the union of all possible values of $next(R)$. From fixed point theory [2], if $next(R)$ is inflationary, i.e, $R \subseteq next(R)$, and if $NEXT(R)$ is monotone in R, then $next(R)$ has a unique least fixed point if it has any finite fixed point at all. In this case, the values of $R1$, ..., Rk will be unique when T terminates.

The nondeterministic function $next(R)$ is inflationary if the incremental operations of R_1, ..., R_k are all *insert*. This claim is still true even when we allow some or all of the incremental operations to be *unify*, if we generalize the relation $\subseteq$ in such a way that $X \subseteq Y$ iff for each x in X, there is a y in Y that is more specific* than or equal to x.

Testing monotonicity is difficult in general. But it is easy to to see that **if all the relational left-terms with local relation names are positive, then $NEXT(R)$ is monotone if $next(R)$ is inflationary**. There is no constraint on external relations, since external relations are not modified.

A transcript is called *positive* if all the relational left-terms with local relation names are positive, and all their incremental operations are either *insert* or *unify*. In the knowledge-base system theory, it has been proved that the relations defined by a system of DATALOG rules has a least fixed point if the rules are stratified [13]. It is not difficult to see that a system of DATALOG rules is stratified only if it can be organized into a set of positive RSL transcripts whose dependency graph is acyclic.

3.4. Expressive Power

In [5], Davis and Weyuker present a small language L which has the expressive power of a Turing machine. The programs in L use only three kinds of instructions: $v := v+1$, $v := v-1$, and **if** $v \neq 0$ **goto** A. In Appendix I, we show that any L program can be simulated by an RSL transcript. For this reason, the termination of an RSL transcript is undecidable in general.

4. Implementation

The main difficulty in the implementation of rule based systems is the tremendous search space. For example, to evaluate the RSL rule

```
parent(.x, .y) and
same_generation(.x, .z) and
parent(.z, .w)
-> same_generation(.y, .w);
```

we have to search the relation *parent* for all the pairs of the tuples [*.x*, *.y*] and [*.z*, *.w*] such that [*.x*, *.z*] is a tuple in the relation *same_generation*. This search has to be done again after each modification of the relations *parent* and *same_generation*, and this repeated search has to be done

* We say y is more specific then x if y is a unification variable, or x and y are lists of the same length, and y is more specific then x componentwise.

for each rule in the transcript.

Our solution is to maintain a set *pool* of partially instantiated rules, called instances. An instance I is a pair (r, B) containing a rule r and a partial variable binding B. The rules can be implemented as pointers to the locations where the rules are actually stored, and the bindings can be implemented as sets of pairs. If the domain of B contains all the pattern variables of r, then we say that the instance I is *ready* and the binding B is *complete*. For correctness, we want *pool* to be large enough to contain all the active instances (recall that these are ready instances whose left-hand-sides evaluate to true). For the efficiency, however, we want *pool* to be as small as possible.

Pool is initialized as follows. First, the rules with a leading *match* predicate are considered. A tree pattern matching [3] is done in linear time to determine the subset of the patterns occurring in the *match* predicates that match some of the subtrees of the parse tree. For each rule with a matched pattern, one instance is created for each matching. No instance is created for those rules with unsatisfied *match* predicates. The rules with no *match* predicates are added to *pool* with an empty binding set.

If there is any external relation, then they are used to initialize *pool* also. Let R be an external relation, and t be a tuple in R. Let $I = (r, B)$ be an instance in *pool*. Let $R(p)$ be a positive left-term of r with the parameter list p. If t matches p with a resulting binding $B_{p,t}$, and $B_{p,t}$ is consistent with the binding B, then a new instance $(\mathrm{r}, B_{t,p} \cup \mathrm{B})$ is inserted into *pool*.

After the initialization, the main loop begins:

```
new := the set of ready instances;
while new ≠ {} loop
  remove an instance $(r, B)$ from new;
  evaluate the left-hand-side of $r$ with the binding $B$;
  if the result is true
  then perform the right-hand-side actions of $r$;
      if any new term $t$ is added to a relation $R$ then
1         for each instance $(r_t, B_t)$ having a positive
2              left-term $R(p)$ such that $t$ matches $p$ with a resulting binding
3              $B_{t,p}$ that is consistent with $B_t$ loop
             if $B_t$ is already complete
             then add $(r_t, B_t)$ to new;
             elseif $B_{t,p} \neq \{\}$ then
                  add the new instance $(r, B_{t,p} \cup B_t)$ to pool;
                  if $B_{t,p} \cup B_t$ is complete
```

```
                    **then** add $(r, B_{t,p} \cup B_t)$ to *new*;
                    **end if**;
                **end if**;
            **end loop**;
        **end if**;
    **end if**;
**end loop**;
```

Indices are created to speed up the search of lines 1, 2, and 3. Since the indexing method for ready instances is different than that for the non-ready instances, we store the two kinds of instances separately. For ready instances, the index is established on the instantiated positive relational left-terms. For non-ready instances, the indices are established in two stages. Let P be the set of non-ready instances in *pool*. If $I = (r, B)$ is an instance, t is a term in r, and t' is obtained from t by variable substitution using B, then we say t' is a term of I. A term of I is *partial* if it still contains variables. Let T be the set of partial positive relational left-terms contained in instances in P. There is one first stage index $INDX\_1_{R,i}$ for each component i of each relation R:

$$INDX\_1_{R,i} = \{\ [x, t]: t \text{ is a term in } T \text{ with predicate symbol } R, \text{ the } i\text{th component of } t \text{ is } x, \text{ and } x \text{ is a constant }\}$$

In addition, for each relation R, there is an index $INDX\_1_{R,0}$ that contains the set of terms in T with no constant components.

The second stage index $INDX\_2$ maps each term in T to the instances in P that contains it:

$$INDX\_2 = \{\ [t, I]: t \in T, I \in P, t \text{ is a positive relational left\_term of } I\ \}$$

Now suppose a new tuple t is added to the relation R whose arity is k. To find the set of instances in P that can be further instantiated with t, we first use $INDX\_1$ to find the R-term in T whose arguments may match t:

$$may\_match = INDX\_1_{R,0} \cup INDX1_{R,1}(t[1]) \ldots \cup INDX\_1_{R,k}(t[k])$$

where $t[i]$ is the ith component of t. For each R-term tr in *may_match*, we check whether its argument list matches t with some nonempty binding B_t. If so, we find the instances in *pool* that contain tr as a positive left-term:

$$affected = INDX\_2\{tr\}$$

For each instance (r, B) in *affected* we create a new instance $(r, B \cup B_t)$, and increment $INDX\_1$ and $INDX\_2$ accordingly.

The following examples give further details of the indexing method. Consider the rule from the transcript in Example 1:

r: *parent*(.*x*, .*y*) **and**

parent(.*z*, .*w*) **and**
same_generation(.*z*, .*x*)
-> *same_generation*(.*y*, .*w*);

Suppose the instance I = (*r*, {[.*x*, $p1$], [.*z*, $p1$]}) is in the *pool* already, and a new tuple t = [$p1$, $p2$] is added to *parent*. We expect that the active instance I_0 = (*r*, {[.*x*, $p1$], [.*y*, $p2$], [.*z*, $p1$], [.*w*, $p2$]}) will be generated. As described above, we use *INDX*_1 to find the two partial terms *parent*($p1$, .*y*) and *parent*($p1$, .*w*), and use *INDX*_2 to find the instance I for further instantiation. By matching *parent*($p1$, .*y*) with *parent*($p1$, $p2$), we create the new instance $I1$ = (*r*, {[.*x*, $p1$], [.*z*, $p1$], [.*y*, $p2$]}). By matching *parent*($p1$, .*w*) with *parent*($p1$, $p2$), we create the new instance $I2$ = (*r*, {[.*x*, $p1$], [.*z*, $p1$], [.*w*, $p2$]}). Indices are modified accordingly: for $I1$, we add the tuple [*parent*($p1$, .*w*), $I1$] to *INDX*_2; for $I2$, we add the tuple [*parent*($p1$, .*y*), $I2$] to *INDX*_2. If we stop here, then the ready instance I_0 is still missing. The actual algorithm continues as follows. The newly added *INDX*_2 entries mean that if a new tuple [$p1$, .*w*] is added to *parent*, than $I1$ can be further instantiated; and if a tuple [$p1$, .*y*] is added to *parent*, than $I2$ can be further instantiated. Since both [$p1$, .*w*] and [$p1$, .*y*] match *t*, the two instances $I1$ and $I2$ are retrieved immediately and the ready instance I_0 is created.

Here is another interesting example. Consider the rule

r: A(.*x*, f(.*y*)) **and** B(.*y*, f(.*x*)) -> C(.*x*, .*y*);

where A and B are two local relations, and f is a system defined function with $f(b) = b$ and $f(a) = c$. Let I = (*r*, {}) be an instance in *pool*. Suppose first the tuple $t1$ = [a, b] is added to A and then the tuple $t2$ = [b, c] added to B. We show how our algorithm works in this situation. When $t1$ is added to A, the term A(.*x*, f(.*y*)) is retrieved using $INDX\_1_{A,0}$, since both .*x* and f(.*y*) are not instantiated. By matching [.*x*, f(.*y*)] with $t1$, we get the binding {[.*x*, a]}. No binding is generated for .y, however, since f(.*y*) is a function application. As a result, a new instance $I1$ = (*r*, { [.*x*, a] }) is created. Since $f(a) = c$, the tuple [c, B(.*y*, c)] is added to $INDX\_1_{B,2}$ and [B(.*y*, c), $I1$] is added to *INDX*_2. When $t2$ is added to B, the term B(.*y*,c) is retrieved through $INDX\_1_{B,2}$, and the instance $I1$ is retrieved through *INDX*_2 with the term B(.*y*, c). By matching $t2$ with B(.*y*, c), a ready instance $I2$ = (*r*, {[.*x*, a], [.*y*, b]}) is generated.

4.1. Rule splitting

Consider the rule

r: A(.*x*) **and**
B(.*y*) **and**
C(.*x*, .*y*)
-> ABC(.*x*, .*y*);

If our algorithm is implemented naively, at least $|A| * |B|$ instances will be generated from *r* alone, which is not desirable. If we split r into the following two rules:

```
r1:   A(.x) and
      BC(.x, .y)
      -> ABC(.x, .y);

r2:   B(.y) and
      C(.x, .y)
      -> BC(.x, .y);
```

then the number instances generated from $r1$ and $r2$ will be $O(|A|+|B|+|C|)$. Finding a good splitting is actually a problem of join optimization [13], and has been studied extensively in the relational database theory.

5. Performance

We did several experiments on the inference engine to see the effectiveness of our implementation strategy. In these experiments, we applied two typical transcripts, one for control flow analysis, the other for live code analysis, to different groups of SETL programs to see how the running time depended on the size of the output relations. Fig. 3 shows the result of applying the transcript for control flow analysis to ten real SETL programs.

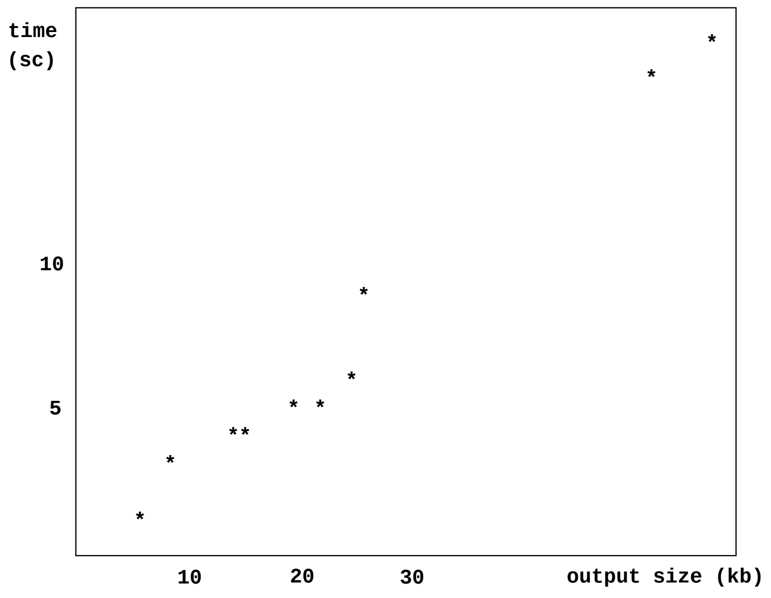

Fig. 3. Control flow analysis on ten real SETL programs

This result does not give a very clear picture of the performance. To reduce the influence of the structures of different programs on the result, we constructed two groups of artificial programs. Each program in the first group contains a sequence of identical **for**-loops, as shown in

Fig.4.

```
program sequence;
   read(x, y);
   for z in y loop
       if z not in x then
           x with:= z;
       end if;
   end loop;
   for z in y loop
       if z not in x then
           x with:= z;
       end if;
   end loop;
    ...
    print(x);
end sequence;
```

Fig. 4. Artificial programs with sequences of loops

In the programs of the second group, these loops are nested, as shown in Fig. 5. The results from the experiments with these artificial programs are shown in Fig. 6, Fig. 7, and Fig. 8.

In control flow analysis, the output is linear to the size of the input program. In live code analysis, the output is quadratic. In both cases, the experiments show that the running time is linear to the size of the output, which is the best we can expect.

6. Comparison with other rule based systems

There are many other implementations of rule based systems. Among those best known are RETE algorithm [6-8, 12] and TREAT algorithm [9-11]. The semi-naive algorithm [13], according to [10], is a special case of TREAT.

In RETE, the left-hand-sides of rules are compiled into a data flow network, as illustrated in Fig. 9. There are two kinds of nodes in the network. The one-input nodes match new tuples with relational left-terms, and the two-input nodes implement the join operations. The output of one-input nodes is stored in the α memories and the output of two-input nodes is stored in the β memories. When a new tuple is inserted into some relation, a sequential search is performed to locate the α memories that need to be modified. For each two-input node, when the memory at one input is modified, a sequential search is performed on the memory on the other input to modify the output β memory. In our system, the search of affected α memories and the

```
 program nested;
      read(x1, y1);
      read(x2, y2);
      ...

      for z1 in y1 loop
          if z1 not in x1 then
                x1 with:= z1;
                for z2 in y2 loop
                    if z2 not in x2 then
                        x2 with:= z2;
                          ...

                    end if;
                end loop;
          end if;
      end loop;
      ...
      print(x2);
      print(x1);
end nested;
```

Fig. 5. Artificial programs with nested loops

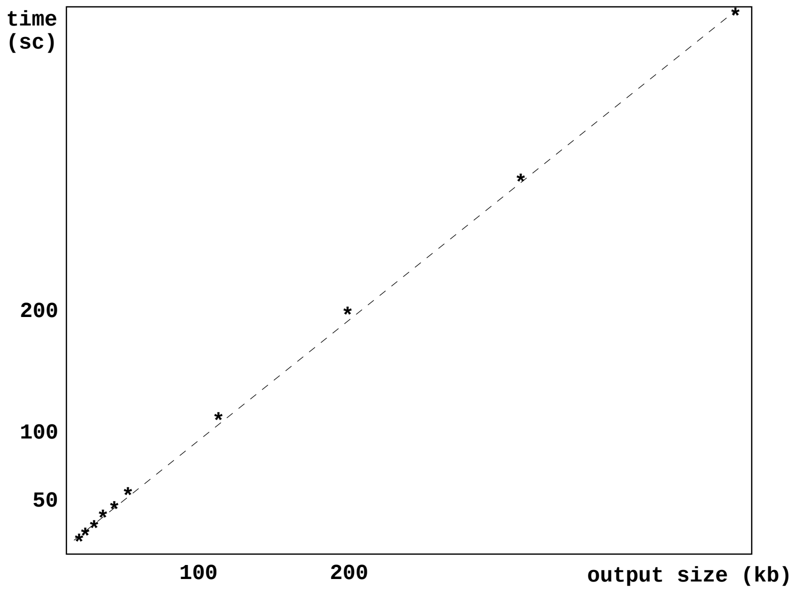

Fig. 6. Live code analysis on sequential loops

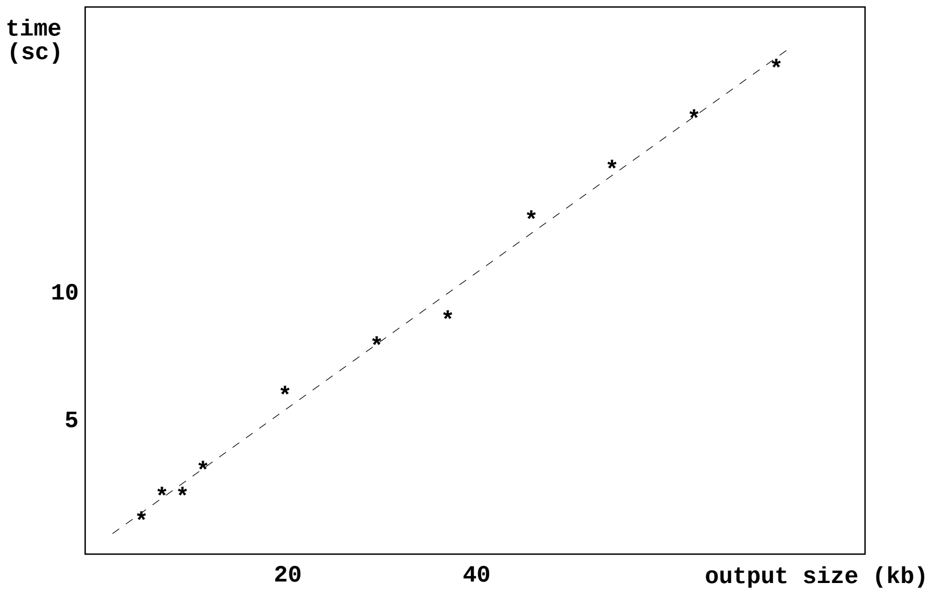

Fig. 7. Control flow analysis on sequential loops

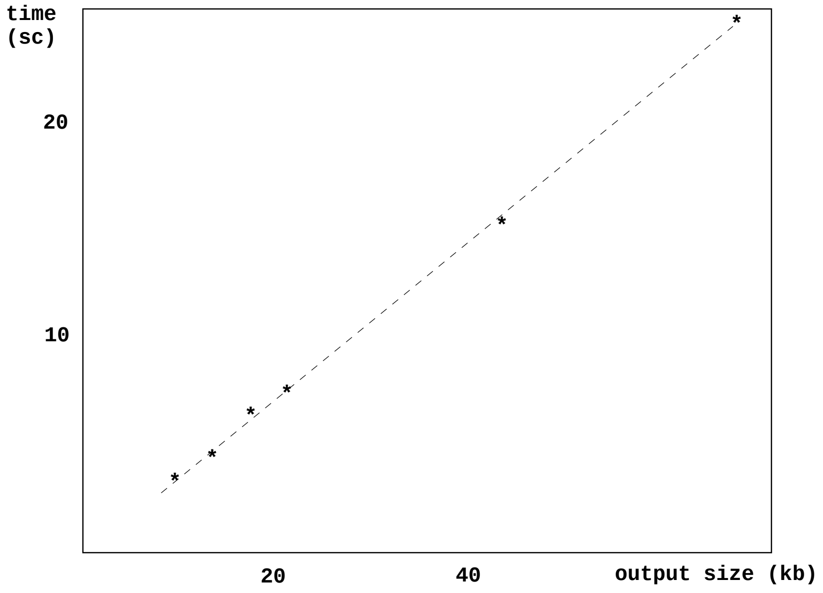

Fig. 8. Control flow analysis on nested loops

incremental modification of the β memories are effectively achieved by maintaining the set of instances, and are performed more efficiently with the two-step indexing.

A close look at our implementation and RETE algorithm reveals that the set of instances in our algorithm corresponds to the union of all the tuples in all the α memories and β memories. Thus the space usage of our implementation is comparable to that of RETE.

The TREAT algorithm saves space by omitting all the β memories. Portions of the β memory elements are computed when needed. Some version of TREAT uses index to speed up

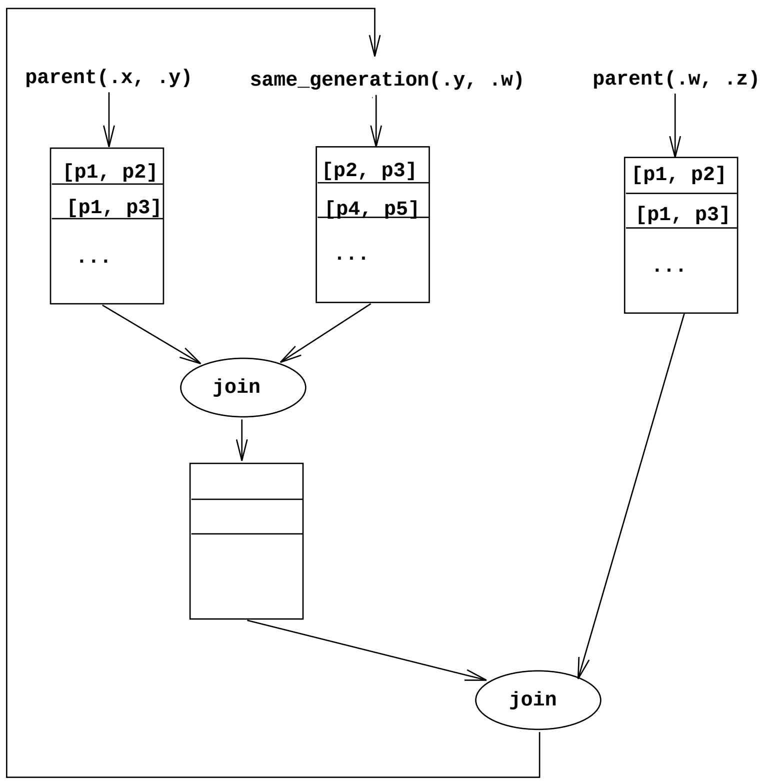

Fig. 9. A RETE network

the search of the α memories in computing the joins, but linear search is still used to locate the α memories that must be modified when new tuples are generated.

Another advantage of our system is that we separate tree patterns and relational patterns. This allows us not only to write semantic rules more concisely and intuitively, but also to implement the language more efficiently using the linear-time bottom-up multi-pattern tree matching algorithm [3].

7. Comparison with attribute grammars

Like RSL, the attribute grammar [1] is also a formal tool of specifying program semantics. In an attribute grammar, semantics are specified as attributes of syntactic objects. While RSL associates semantic rules with tree patterns, the attribute grammar associates semantic rules with grammar productions. Therefore the attribute grammar is more grammar dependent than RSL.

In the attribute grammar, semantic information is strictly propagated along the parse trees. Considering the efficiency of implementation, most systems using attribute grammars restrict the order in which the attributes can be evaluated. For example, in order to evaluate the attributes in one pass through the parse tree, a compiler system usually requires that the computation of an attribute at one node of the parse tree should not use information from its right siblings and the right siblings of its ancestors. With this restriction, it is very difficult to specify the semantic rules in our Example 2 using the attribute grammar. In RSL, however, the parse tree is traversed once only to get the variable bindings, and then the semantic information is computed as database relations. The user do not have to worry about the evaluation order.

Appendix I: Simulation of the Language *L*

We show how to use a RSL transcript to simulate a program written in the language L described in [5], which can express all the partially computable functions.

Language L has three kinds of statements: $v := v+1$, $v := v-1$, and **if** $v \neq 0$ **goto** A.

Consider a program P in L with statements s_1, ..., s_k. To simulate P in RSL, we create a transcript T_P with the following declarations:

```
transcript T_P;
rel next: [string, string];
    value: [string, list];
    done: [string];
key value: [1];
incremental value: replace;
begin
...
end;
```

The transcript body contains the following.

1. A rule to simulate the entrance of the program P:

```
true -> next(s_1, newatom(s));
```

where *newatom*(x) is a system function returning a fresh new string with the prefix x each time it is called.

2. Rules to initialize the variables, one rule for each variable v in P:

```
true -> value(v, []);
```

We use [] to encode 0. By convention, the initial value of each variable is 0.

3. Rules to simulate statements in P. If a statement s_i has the form $v := v+1$, then it is simulated by the rule

```
next(s_i, .x) and
not done(.x) and
value(v, .y)
-> done(.x) and
```

```
    *next*$(s_{i+1}$, *newatom*$(s))$ **and**
    *value*$(v$, $[.y])$;
```

Here $[x]$ encodes v+1 if x encodes v. If s_i has the form $v := v-1$, then it is simulated by two rules:

```
*next*$(s_i, .x)$ **and**
**not** *done*$(.x)$ **and**
*value*$(v$, $[.y])$
-> *done*$(.x)$ **and**
    *next*$(s_{i+1}$, *newatom*$(s))$ **and**
    *value*$(v$, $.y)$;

*next*$(s_i, .x)$ **and**
**not** *done*$(.x)$ **and**
*value*$(v$, $[])$
-> *done*$(.x)$ **and**
    *next*$(s_{i+1}$, *newatom*$(s))$;
```

By convention in L, $0 - 1 = 0$. If s_i has the form **if** $v \neq 0$ **goto** s_j, then it is simulated by two rules:

```
*next*$(s_i, .x)$ **and**
**not** *done*$(.x)$ **and**
*value*$(v$, $[.y])$
-> *done*$(.x)$ **and**
    *next*$(s_j$, *newatom*$(s))$;

*next*$(s_i, .x)$ **and**
**not** *done*$(.x)$ **and**
*value*$(v$, $[])$
-> *done*$(.x)$ **and**
    *next*$(s_{i+1}$, *newatom*$(s))$;
```

The proof of the correctness should be straight forward.

Acknowledgment I want to thank Robert Paige for his encouragement and support, without which the design and implementation of RSL would be impossible.

References

1. Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

2. Cai, J. and Paige, R., ‘‘Program Derivation by Fixed Point Computation,’’ *Science of Computer Programming*, vol. 11, no. 3, pp. 197-261, Apr. 1989.

3. Cai, J., Paige, R., and Tarjan, R., ‘‘More Efficient Bottom Up Tree Pattern Matching,’’ in *Proc. CAAP 90*, ed. A. Arnold, Lecture Notes in Computer Science, vol. 431, pp. 72-86, Springer-Verlag, 1990.

4. Cai, J. and Paige, R., *Annul Technical Report, Grant AFOSR-91-0308, September 1, 1991 to August 31, 1992*, Department of Computer Science, New York University/Courant Institute, 1993.

5. Davis, M. and Weyuker, E., *Computability, Complexity, and Languages*, Academic Press, 1983.

6. Forgy, C. L., ‘‘Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem,’’ *Artificial Intelligence*, no. 19, pp. 17-37, North-Holland, 1982.

7. Lee, H. S. and Schor, M. I., ‘‘Dynamic augmentation of generalized Rete networks for demand-driven matching and rule updatings,’’ in *Proceedings 6th IEEE Conference on Artificial Intelligence Applications*, pp. 123-129, 1990.

8. Lee, H. S. and Schor, M. I., ‘‘Match algorithms for generalized rete networks,’’ *Artificial Intelligence*, no. 54, pp. 249-274, 1992.

9. Miranker, D., ‘‘TREAT: a better match algorithm for AI production systems,’’ in *Proceedings AAAI-87*, pp. 42-47, Seattle, WA, 1987.

10. Miranker, D. P. and Lofaso, B. J., ‘‘The organization and performance of a TREAT-based production system compiler,’’ *IEEE Trans. Knowl. Data Eng.*, no. 3(1), pp. 3-10, 1991.

11. Miranker, D. P., Lofaso, B. J., Farmer, G., Chandra, A., and Brant, D., ‘‘On a Treat based production system compiler,’’ in *Proc. 10th int. Conf. Expert Syst.*, pp. 617-630, Avignon, France, Jun 1990.

12. Schor, M. I., Daly, T., Lee, H. S., and Tibbitts, B., ‘‘Advances in Rete pattern matching,’’ in *Proceedings AAAI-86*, pp. 226-232, Philadelphia, PA, 1986.

13. Ullman, J., *Principles of Database and Knowledge-Base Systems*, I, Computer Science Press, 1988.