数理解析研究所講究録
第 396 巻 1980 年 39-63

A Method for Synthesis of Data Base Access Programs

Akinori Yonezawa

Dept. of Information Science
Tokyo Institute of Technology
Oh-okayama, Meguro Tokyo, Japan

Abstract

A method is presented for synthesizing programs which answer to relational database queries expressed in a language based on predicate calculus. Given a query at the conceptual level and the description of physical representations of relations (or predicates), a program is synthesized which manipulates stored data at the physical (internal) level. The method of synthesis is based on successive transformations of queries by applying "rules" which express procedural interpretations of logical formulas. Queries expressed by recursively defined relations (formulas) are successfully handled by our method. The target language in which synthesized programs are written is required to support functional arguments.

Keywords program synthesis, relational database, database query, predicate calculus, recursively defined relation, procedural interpretation of predicates, functional argument

§1 Introduction

Program synthesis is the systematic derivation of a program from a specification [MW79]. Most research activities in program synthesis take approaches based on theorem proving techniques. In these approaches, a program specification is given as a theorem in some theory and proving the theorem is first attempted. If a successful proof for the theorem is found, a program is synthesized by systematically extracting information from the proof. A variety of techniques have been developed corresponding to the variety of underlying logic systems; Examples are [G69], [WL69] based on resolution-type proof procedures, [HT79] based on Gentzen's Natural Deduction System and [S79] based on Gödel's interpretation.

The deductive approach taken by Reiter [R78] and Chang [C78] in relational data bases can be viewed as synthesis of data base access programs in which a query is considered as a program specification. Their approach is also based on theorem proving techniques.

The present paper proposes a new approach to program synthesis in the domain of relational data bases. Our method is based on successive transformations of queries by a set of "rules" which express <u>procedural</u> interpretations of logical formulas in which queries are stated. Given a query and the description of physical implementations of relations (appearing in the query), both of which are written in a first order language, a program which actually manipulates physically stored data is synthesized.

We start with an informal explanation of our method.

§2 A Simple Example

To illustrate our program synthesis method, let us consider the following situation. A relation ROOM(Sect,Num) which associates a room number with the section name it belongs to is defined at the conceptual level[AN75] in a data base. Suppose a user issues a query Q to know the set of room numbers which belong to the INFO section. Regarding the relation ROOM(s,n) as a two place predicate (which gives the truth value of a statement that a room number n belongs to a section s), this query is expressed in a familiar notation of mathematics as follows:

$$Q = \{n \mid \mathrm{ROOM}(\mathrm{INFO},n)\}$$

What the user wants is an actual enumeration of the elements of the set specified by this notation. To obtain such an enumeration, we must first know how the relation (defined at the conceptual level) is actually implemented at the physical (or internal) level and then we must compose a data base access program which does this enumeration by manipulating physically stored data.

To describe how such a relation is implemented at the physical level, we use a language of first order logic. Suppose the ROOM relation viewed as a predicate is implemented as a list of pairs each of which consists of a section name and a room number. This can be expressed as:

$$\mathrm{ROOM}(s,n) \equiv \exists x(\ x \in \mathrm{ROOML} \wedge \mathrm{CAR}(x)=s \wedge \mathrm{CDR}(x)=n\ ),$$

where ROOML is a constant list, s and n are free variables. In our language, $e \in l$ is an abbreviation of a two place predicate which asserts that e is an element of a list l. (The definition of the list membership and an axiomatization of lists are given in [CT77].) CAR and CDR are one place functions which give the first part and the second part of a pair, respectively. We call this kind of logical equivalence which expresses the internal representation of a relation, a <u>physical repre-</u>

sentation definition (PRD) of the relation.

Substituting this PRD, the query Q is transformed into:

$$\{n \mid \exists x(x \leftarrow ROOML \wedge CAR(x)=INFO \wedge CDR(x)=n)\}$$

and by equality substitution, we get:

$$\{CDR(x) \mid \exists x(x \leftarrow ROOML \wedge CAR(x)=INFO)\} \qquad (\&)$$

This is logically equivalent to:

$$\{CDR(\alpha) \mid \bigvee_{\alpha \leftarrow ROOML} (CAR(\alpha)=INFO)\} \qquad (\&\&)$$

where α is an index variable ranging over the set whose elements are the same as those of ROOML. Furthermore, this set is equivalent to

$$\bigcup_{\alpha \leftarrow ROOML} \{CDR(\alpha) \mid CAR(\alpha)=INFO\} \qquad (\&\&\&)$$

The set expressed by (&&&) is a collection of the second (cdr-) part of α which is an element of the list ROOML and α's first (car-) part is INFO. This suggests a procedural interpretation of the set notation (&&&), or a program which enumerates all the elements of the set. The following program is an example of such an interpretation.

```
S ← φ; L ← ROOML;
while L ≠ nil do
  begin α ← head(L);
        if CAR(α) = INFO then S ← S ∪ {CDR(α)} ;        (P1)
        L ← tail(L)
  end;
return(S)
```

This program is, in turn, a procedural interpretation of the set expressed by the original query Q.

We have been following somewhat tedious logical steps to obtain a program for the query. These steps can be short-cut and generalized by few types of transformation rules. By noting that the existentially quantified variable x in (&) behaves as an index which ranges over the list elements of ROOML, we adopt an $\exists$-quantifier elimination rule of the following form.

(ES1a) $\{s(x) \mid \exists x(P(x) \wedge x \propto L)\} \Rightarrow \{s(\alpha) \mid P(\alpha) \wedge \alpha \propto L\}$

where α is a newly generated variable, s(x) is a term which is constructed from variable x, and L is a set or list. $\propto$ stands for the set membership or the list membership. Applying this rule to (&) with P(x) being CAR(x)=INFO and s(x) being CDR(x), we obtain:

$$\{CDR(\alpha) \mid CAR(\alpha)=INFO \wedge \alpha \propto ROOML\} \qquad (\%)$$

We can (procedurally) interpret this notation in the same way as (&&&) and may obtain the program (P1).

To generalize a codification process, we introduce an operator pr which transforms a formula or set notation $\mathcal{F}$ into a program that gives a procedural interpretation of $\mathcal{F}$. The above codification process (from (%) to (P1)) is an application of the following rule.

(CRS1) $\underline{pr}(\{s(x) \mid x \propto L \wedge P(x)\}) \Rightarrow$ generate[$\underline{pr}$(L); λx.$\underline{pr}$(s(x)); λx.$\underline{pr}$(P(x))]

generate[l; c; p] is a procedure which takes a value argument l, and two functional arguments c and p. This procedure returns a list of distinct items each of which is obtained by applying a function c to each element (in a list l) which satisfies a condition (predicate) p. λx.<body> is a notation for a procedure definition declaring x as the formal argument and <body> as the procedure body. Thus the right-hand side of (CRS1) denotes an invocation of "generate" with the following three parameters:

- a list pr(L) which is the result of application of pr to L,
- a procedure λx.pr(s(x)) whose formal parameter is x and body, pr(s(x)),
- a procedure λx.pr(P(x)) whose formal parameter is x and body, pr(P(x)).

Applying this rule (CRS1) to (%) with the instantiations of P(x) by CAR(α)=INFO, L by ROOML, and s(x) by CDR(α) and using other simple codification rules, the following program (written in a fashion of the LISP meta-expression [M65]) is obtained.

```
generate[ROOML; λx.cdr[x]; λx.equal[car[x]; INFO]]
```

Note that cdr, car, equal etc.(written in small letters) denote procedures defined in LISP. They must be distinguished from functions CAR, CDR, = which are used in our logical language. We deliberately leave an actual procedure body of "generate" unspecified(+) because many implementations are possible in many programming languages as long as they allow procedures as parameters and support list structures.

§3 Logical Language and Query Language

As illustrated in the previous section, both database queries and physical representations of relations (defined at the conceptual level) are expressed in terms of a logical language. To present our program synthesis method in a formal fashion and make its scope clear, we first discuss our logical language.

Roughly speaking, the language is an extension of first order predicate calculus with equality, set membership and list membership.

(term)

1. An individual constant (written in capital letters) or variable (written in small letters) is a term.
2. If f is an n-place function symbol (written in capital letters) and t1,..,tn are terms, then f(t1,..,tn) is a term.
3. All terms are constructed through 1 and 2.

We assume that list processing functions such as CAR, CDR, CONS, LIST and an identity function ID, and a constant NIL are included in the language. Below we use a special symbol $\propto$ to stand for ε (set membership) or ϵ (list membership).

(formula)

1. If t is a term and l is a set or list, then $t \propto l$ is a formula.
2. If t1 and t2 are terms, t1=t2 is a formula.
3. If P is an n-place predicate symbol (written in capital letters) and t1,...tn are terms, then P(t1,..,tn) is a formula.
4. If A and B are formulas, then $\neg A$, $A \vee B$, $A \wedge B$ and $A \rightarrow B$ are formulas.
5. If A is a formula, x is a variable and l is a set or list, then $\forall x A$, $\exists x A$, $\bigvee_{x \propto l} A$ and $\bigwedge_{x \propto l} A$ are formulas.
6. All formulas are constructed through 1 to 5.

+) An example of the body of "generate" in LISP is:

```
generate[l; c; p]
      = nd[cond[null[l] → NIL;
               p[car[l]] →  cons[c[car[l]];
                                 generate[cdr[l]; c; p]];
               T → generate[cdr[l]; c; p]]]
```

where nd[l] eliminates repetitive elements of a list l.

An n-place relation defined in the data base is considered as an n-place predicate and user-defined predicates can always be included in the language.

(set)

1. $\{\}$ or NIL is a set which denotes an empty set.
2. If t1,..,tn are distinct terms, $\{t1,...,tn\}$ is a set whose elements are t1,..,tn.
3. If t(x) is a term containing a unique variable x and P(x) is a formula with free(+) occurrences of x, then $\{t(x) \mid P(x)\}$ is a set whose elements are distinct terms t(x) such that x satisfies P(x). When t(x) is a simple variable x, t is considered as an identity function ID.
4. If S1 and S2 are sets, then $S1 \cap S2$ (intersection), $S1 \cup S2$ (union), and S1 - S2 (difference) are sets.
5. If S(x) is a set notation containing a free variable x and l is a set or list, then $\bigcup_{x \propto l} S(x)$ (indexed union) and $\bigcap_{x \propto l} S(x)$ (indexed intersection) are sets.
6. All sets are constructed through 1 to 5.

The restriction that only a single variable is allowed in the term t(x) of the definition 3 is not essential; we can easily extend our method to accomodate more than one variable.

The queries we allow are classified into two types: a set-type (or open [GMN78]) queries and formula-type (or closed) queries.

(query)

1. A set-type query is of the form $\{t(x) \mid P(x)\}$ defined in 3 of the set definition above and it requests the system to list up all the elements.
2. A formula-type query is a formula defined above and requests the system to return the truth value (yes or no) of the formula evaluated in the data base.

Since the definition of queries above allows very general formulas in first order predicate calculus, a general program synthesis method should include an algorithm for a universal theorem proving of such formulas. But our synthesis method restricts itself to queries which

+) x can be a bound variable if it is existentially quantified.

can be answered by searching physically stored data.[+].

§4 Logical Transformation Rules

Our method of program synthesis is based on the successive transformation of queries by two kinds of rules: logical transformation rules and codification rules. We shall discuss the former ones in this section.

As the example in §2 suggests, general algorithmic structures of synthesized programs are summerized as " Given a source of data items, check each data item one by one as to whether it satisfies some conditions and if it meets the conditions, do something on the item or report the truth value of the conditions." This general algorithm is usually used in a nested manner. ROOML (in the example in §2) is such a source of data items. Successful synthesis of efficient programs depends upon the discovery and choice of data items sources which may be implicit in formulas.

The implicit source are often found through quantifiers in formulas. Simple examples are set notations and formulas of the following forms.

$$\{s(x) \mid \exists y(R(x,y) \wedge y \propto L)\} \quad , \quad \exists z(P(z) \wedge z \propto L)$$

where s(x) is a term and L is a set or list . Queries of these forms are evaluated by instantiating y or z with each element of L one by one. Thus these existentially quantified variables behave like index variables ranging over L. To express this we employ the following rules. (Index variables should not have name conflicts with other variables.)

(ES2a)	$\{s(x) \mid \exists y(R(x,y) \wedge y \propto L)\}$	$\Rightarrow$	$\{s(x) \mid R(x,\alpha) \wedge \alpha \propto L\}$
(EL1a)	$\exists z(P(z) \wedge z \propto L)$	$\Rightarrow$	$\bigvee_{\alpha \propto L} P(\alpha)$

L is the source of data items in these rules. If such explicit sources are not found, which is a more general case, we have the following rules.

+) We can, if necessary, employ any of theorem proving techniques that have been developed by many researchers.

(ES2)	$\{s(x) \mid \exists y(R(x,y) \wedge Q(y))\}$	$\Rightarrow$	$\{s(x) \mid R(x,\alpha) \wedge \alpha \in \{y \mid Q(y)\}\}$
(EL1)	$\exists z(P(z) \wedge Q(z))$	$\Rightarrow$	$\bigvee_{\alpha \in \{z \mid Q(z)\}} P(\alpha)$

An explicit set (or list) L is replaced by the set specified by a predicate Q. Sometimes there are cases where the sources are hard to find; for example,

$$\{s(x) \mid \exists y(R(x,y) \wedge S(x,y)\} \text{ , } \{s(x) \mid \exists y A(x,y)\} \text{ , } \exists z B(z)$$

where A(x,y), $R(x,y) \wedge S(x,y)$ and B(x) cannot break intó the patterns of (ES2) or (EL1). In such cases, we must introduce the domains of quantified variables to create the sources of data items. This is expressed as:

(ES3)	$\{s(x) \mid \exists y(R(x,y) \wedge S(x,y))\}$	$\Rightarrow$	$\{s(x) \mid x \in \text{domain}(R.1) \wedge \bigvee_{\alpha \in \{y \mid R(x,y)\}} S(x,\alpha)\}$
(ES4)	$\{s(x) \mid \exists y A(x,y)\}$	$\Rightarrow$	$\{s(x) \mid A(x,\alpha) \wedge \alpha \in \text{domain}(A.2)\}$
(EL2)	$\exists z B(z)$	$\Rightarrow$	$\bigvee_{\alpha \in \text{domain}(B.1)} B(\alpha)$

domain(T.n) denotes the domain of a variable in the n-th place of a predicate (relation) T.

The procedural interpretation of (ES3) is that the set is obtained by selecting x (from domain(R.1)) by the condition $\bigvee_{\alpha \in \{y \mid R(x,y)\}} S(x,\alpha)$ and applying s to such x.

For formulas containing universal quantifiers, similar rules are adopted by the same motivation. A list of transformation rules for quantifiers is shown in Figure 1. Notice that $\wedge$ (and) in the case of $\exists$-qunatifiers is replaced by $\rightarrow$ in the $\forall$-quantifier case. Rules for other logical symbols and simplification rules for set notations are given in Appendix. Besides these rules, we use the rules for replacement of logically equivalent formulas, for renaming of bound variables, for equality substitution (ESU) and for definition substitution (DSU).

Universal Quantifier

(AS1) $\{s(x) \mid \forall z(P(z) \rightarrow R(x,z))\} \Rightarrow \{s(x) \mid \bigwedge_{\alpha \varepsilon \{z \mid P(z)\}} R(x,\alpha)\}$

(--a) $\{s(x) \mid \forall z(z \propto L \rightarrow R(x,z))\} \Rightarrow \{s(x) \mid \bigwedge_{\alpha \propto L} R(x,\alpha)\}$

(AS2) $\{s(x) \mid \forall z(R(x,z) \rightarrow S(x,z))\} \Rightarrow$
$\{s(x) \mid x \ \varepsilon \ \text{domain}(R.1) \wedge \bigwedge_{\beta \varepsilon \{z \mid R(x,z)\}} S(x,\beta)\}$

(AS3) $\{s(x) \mid \forall z A(x,z)\} \Rightarrow \{s(x) \mid x \ \varepsilon \ \text{domain}(A.1) \wedge \bigwedge_{\alpha \varepsilon \text{domain}(A.2)} A(x,\alpha)\}$

(AL1) $\forall z B(z) \Rightarrow \bigwedge_{\alpha \varepsilon \text{domain}(B.1)} B(\alpha)$

(AL2) $\forall z(P(z) \rightarrow Q(z)) \Rightarrow \bigwedge_{\alpha \varepsilon \{z \mid P(z)\}} Q(\alpha)$

Existential Quantifier

(ES1) $\{s(x) \mid \exists x(P(x) \wedge Q(x))\} \Rightarrow \{s(\alpha) \mid P(\alpha) \wedge \alpha \varepsilon \{x \mid Q(x)\}\}$

(--a) $\{s(x) \mid \exists x(P(x) \wedge x \propto L)\} \Rightarrow \{s(\alpha) \mid P(\alpha) \wedge \alpha \propto L\}$

(ES2) $\{s(x) \mid \exists y(R(x,y) \wedge Q(y))\} \Rightarrow \{s(x) \mid R(x,\alpha) \wedge \alpha \varepsilon \{y \mid Q(y)\}\}$

(--a) $\{s(x) \mid \exists y(R(x,y) \wedge y \propto L)\} \Rightarrow \{s(x) \mid R(x,\alpha) \wedge \alpha \propto L\}$

(ES3) $\{s(x) \mid \exists y(R(x,y) \wedge S(x,y))\} \Rightarrow$
$\{s(x) \mid x \varepsilon \text{domain}(R.1) \wedge \bigvee_{\alpha \varepsilon \{y \mid R(x,y)\}} S(x,\alpha)\}$

(ES4) $\{s(x) \mid \exists y A(x,y)\} \Rightarrow \{s(x) \mid A(x,\alpha) \wedge \alpha \ \varepsilon \ \text{domain}(A.2)\}$

(--a) $\{s(x) \mid \exists x B(x)\} \Rightarrow \{s(\alpha) \mid B(\alpha) \wedge \alpha \ \varepsilon \ \text{domain}(B.1)\}$

(EL1) $\exists z(P(z) \wedge Q(z)) \Rightarrow \bigvee_{\alpha \varepsilon \{z \mid P(z)\}} Q(\alpha)$

(--a) $\exists z(P(z) \wedge z \propto L) \Rightarrow \bigvee_{\alpha \propto L} P(\alpha)$

(EL2) $\exists z B(z) \Rightarrow \bigvee_{\alpha \ \varepsilon \ \text{domain}(B.1)} B(\alpha)$

(EL3) $\exists z(t=z \wedge P(z)) \Rightarrow P(t)$

Figure 1 Transformation Rules for Quantifiers

§5 Codification Rules

To obtain a procedural interpretation (i.e., program) for a set notation or formula, we use the codification rules listed in Figure 2. Most rules are expressed by recursive applications of a codification operator pr. Namely, an application of pr on a set notation or formula results in a procedure (in the target language) whose arguments contain pr operator applications. Though procedures (or procedure invocation, more precisely) in the rules are written in a style of LISP meta-expressions, actual implementations of the procedures may be written in other languages as long as they have functional argument facility and support list processing capability.

The informal meaning of each procedure is stated below. Note that in i-union and i-intersect, each element of a list l is used as an index for generalized union and intersection operations, and f is a function which generates a list for a given index value.

- generate[l; c; p] : list up items (without repetition) each of which is obtained by applying a function c to each element of l which satisfies a predicate p.
- some-meet?[l; p] : ask whether or not some element in a list l satisfies a predicate p.
- all-meet?[l; p] : ask whether or not all the elements in a list l satisfy a predicate p.
- i-union[l; f] : return a list whose elements are the union of sets which are obtained by applying a function f to each element in a list l.
- i-intersect[l; f] : return a list whose elements are the intersection of sets which are obtained by applying a function f to each element in a list l.

<set>

(CRS0) $pr(\{x \mid x \propto L\}) \Rightarrow pr(L)$

(CRS1) $pr(\{s(x) \mid x \propto L \wedge P(x)\})) \Rightarrow$ $generate[pr(L);\ \lambda x.pr(s(x));\ \lambda x.pr(P(x))]$

(--a) $pr(\{s(x) \mid x \propto L\}) \Rightarrow generate[pr(L);\ \lambda x.pr(s(x))\ ;\ TRUE]$

(CRS2) $pr(\{s(t) \mid \bigvee_{t \propto L} P(t)\}) \Rightarrow generate[pr(L);\ \lambda x.pr(s(x))\ ;\ \lambda x.pr(P(x))]$

(CRS3) $pr(\bigcup_{t \propto L} S(t)) \Rightarrow i\text{-}union[pr(L);\ \lambda t.pr(S(t))]$

(--a) $pr(\{s(x) \mid P(x,t) \wedge t \propto L\}) \Rightarrow i\text{-}union[pr(L);\ \lambda t.pr(\{s(x) \mid P(x,t)\})]$

(CRS4) $pr(\bigcap_{t \propto L} S(t)) \Rightarrow i\text{-}intersect[pr(L);\ \lambda t.pr(S(t))]$

(CRS5) $pr(S \cup T) \Rightarrow union[pr(S);\ pr(T)]$

(CRS6) $pr(S \cap T) \Rightarrow intersect[pr(S);\ pr(T)]$

(CRS7) $pr(S - T) \Rightarrow delete[pr(S);\ pr(T)]$

(CRS8) $pr(\{s(t)\}) \Rightarrow list[pr(s(t))]$

(CRS9) $pr(\{t1, \ldots, tn\}) \Rightarrow list[pr(t1), \ldots, pr(t2)]$

<formula>

(CRL0) $pr(s(t) \propto L) \Rightarrow some\text{-}meet?[pr(L);\ \lambda x.equal[x;\ pr(s(t))]]$

(CRL1) $pr(\bigvee_{t \propto L} P(t)) \Rightarrow some\text{-}meet?[pr(L);\ \lambda t.pr(P(t))]$

(CRL2) $pr(\bigwedge_{t \propto L} P(t)) \Rightarrow all\text{-}meet?[pr(L);\ \lambda t.pr(P(t))]$

(CRL3) $pr(\neg p) \Rightarrow not\ pr(p)$ (CRL4) $pr(p \vee q) \Rightarrow or[pr(p);\ pr(q)]$

(CRL5) $pr(p \wedge q) \Rightarrow and[pr(p);\ pr(q)]$

<term>

(CRT1) $pr(CAR(t)) \Rightarrow car[pr(t)]$ (CRT2) $pr(CDR(t)) \Rightarrow cdr[pr(t)]$

(CRT3) $pr(CONS(t1,t2)) \Rightarrow cons[pr(t1);\ pr(t2)]$

(CRT4) $pr(t1 = t2) \Rightarrow equal[pr(t1);\ pr(t2)]$

(CRT5) $pr(t = NIL) \Rightarrow null[pr(t)]$ (CRT6) $pr(ID(t)) \Rightarrow id$

(CRT7) $pr(<constant\text{-}or\text{-}variable>) \Rightarrow <constant\text{-}or\text{-}variable>$

Figure 2 Codification Rules

§6 Recursive Relations and Recursive Programs

To accomodate a wide variety of queries, it is often necessary to use relations which are not stored explicitly in the data base, but are logically derivable from explicitly stored relations. (This kind of relation or a set of relations may be called a "view" in the data base literature [D75].) When a relation is defined solely in terms of explicitly stored relations, no complications arise and we can simply substitute the derived relation by its definition. But a consideration is needed when the definition contains itself, namely the relation is defined recursively. Such an example is ABOVE(x,y) relation (a block x is above a block y) which is derived from ON(x,y) relation (a block x is on a block y) where the ON relation is assumed to be explicitly stored. The definition is:

$$\mathrm{ABOVE}(a,b) \equiv \mathrm{ON}(a,b) \vee \exists z(\mathrm{ON}(a,z) \wedge \mathrm{ABOVE}(z,b)).$$

(Assume that free variables a and b are universally quantified.)

Using this ABOVE relation, we can issue a query

$$Q = \{e \mid \mathrm{ABOVE}(A,e)\}$$

which requests to return a set of all the blocks below a specfied block A. To synthesize a program for this query, first we substitute the above definition for the formula in the query (after appropriate instantiations of free variables $(a \leftarrow A,\ b \leftarrow e)$).

$$\Rightarrow \{e \mid \mathrm{ON}(A,e) \vee \exists z(\mathrm{ON}(A,z) \wedge \mathrm{ABOVE}(z,e))\} \qquad (\S)$$

Our strategy to deal with the recursion is to leave the recursive occurence of ABOVE(z,e) in the formula intact and substitute a physical representation definition of ON for the two occurences of ON. Suppose the ON relation is implemented as a list ONL of pairs where the first part of each pair represents a block on top of a block represented by the second part. This is expressed as:

$$ON(a,b) \equiv \exists x(x \in ONL \wedge CAR(x)=a \wedge CDR(x)=b)$$

Substituting the right-hand side for ON in (§), we obtain:

$$\Rightarrow \left\{ e \,\middle|\, \begin{array}{l} \exists x(x \in ONL \wedge CAR(x)=A \wedge CDR(x)=e) \vee \\ \exists z(\exists x(x \in ONL \wedge CAR(x)=A \wedge CDR(x)=z) \wedge ABOVE(z,e)) \end{array} \right\}$$

Factoring common expressions by the laws of change in the order and scope of quantifiers, and the distribution law[(+)] of an existential quantifier and $\vee$, the query is simplified into:

$$\Rightarrow \{e \mid \exists x\,[(x \in ONL \wedge CAR(x)=A) \wedge (CDR(x)=e \vee \exists z(CDR(x)=z \wedge ABOVE(z,e)))]\}$$

Several applications of logical transformarion rules are in order.

$$\Rightarrow \{e \mid (CDR(\alpha)=e \vee \exists z(CDR(\alpha)=z \wedge ABOVE(z,e))) \wedge \alpha \varepsilon \{x \mid x \in ONL \wedge CAR(x)=A\}\} \quad \text{by (ES2)}$$

$$\Rightarrow \{e \mid (CDR(\alpha)=e \vee ABOVE(CDR(\alpha),e)) \wedge \alpha \varepsilon \{x \mid x \in ONL \wedge CAR(x)=A\}\} \quad \text{by (EL3)}$$

$$\Rightarrow \bigcup_{\alpha \varepsilon \{x \mid x \in ONL \wedge CAR(x)=A\}} \{e \mid CDR(\alpha)=e \vee ABOVE(CDR(\alpha),e)\} \quad \text{by (SE3) in Appendix}$$

$$== \bigcup_{\alpha \varepsilon \{x \mid x \in ONL \wedge CAR(x)=A\}} (\{CDR(\alpha)\} \cup \{e \mid ABOVE(CDR(\alpha),e)\}) \quad \text{by (SE2),(SE7a) in Appendix}$$

Now we use codification rules.

$$\underline{pr}(\{e \mid ABOVE(A,e)\}) \Rightarrow \underline{pr}\left(\bigcup_{\alpha \varepsilon \{x \mid x \in ONL \wedge CAR(x)=A\}} \{CDR(\alpha)\} \cup \{e \mid ABOVE(CDR(\alpha),e)\}\right)$$

$$\Rightarrow \text{i-union}[\underline{pr}(\{x \mid x \in ONL \wedge CAR(x)=A\}); \lambda\alpha.\underline{pr}(\{CDR(\alpha)\} \cup \{e \mid ABOVE(CDR(\alpha),e)\})] \quad \text{by (CRS3)}$$

$$\begin{array}{l} == \text{i-union}[\text{generate}[ONL;\ id;\ \lambda x.\text{equal}[\text{car}[x];\ A]]; \\ \qquad \lambda\alpha.\text{union}[\text{list}[\text{cdr}[\alpha]];\ \underline{pr}(\{e \mid ABOVE(CDR(\alpha),e)\})]] \quad (\S\S) \end{array}$$

by (CRS1, CRT1, CRT4, CRT7, CRS5, CRT2)

Here we wish to eliminate the pr operator applied to $\{e \mid ABOVE(\ldots)\}$. To do so, we need the following special codification rule which introduces recursive invocations of the program to be synthesized.

(+) $\exists x P(x) \vee \exists x Q(x) \equiv \exists x(P(x) \vee Q(x))$

Let Q(a) be a query containing a free variable a. When the result of codification of Q(a) contains occurrences of pr(Q(t)) where t is a term, namely,

if pr(Q(a)) becomes $\mathcal{P}$(pr(Q(t))) ,
then pr(Q(a)) = f[a] where λx.f[x] = $\mathcal{P}$(f[pr(t)]) (+)

Applying this rule to (§§), we obtain a program for the original query.

pr({e|ABOVE(A,e)}) = above[A]

where

above[z] = i-union[generate[ONL; id; λx.equal[car[x]; z]]; $\lambda\alpha$.union[list[cdr[α]]; above[cdr[α]]]]

Note that if for some x, above(x,x) holds, the synthesized program does not terminate.

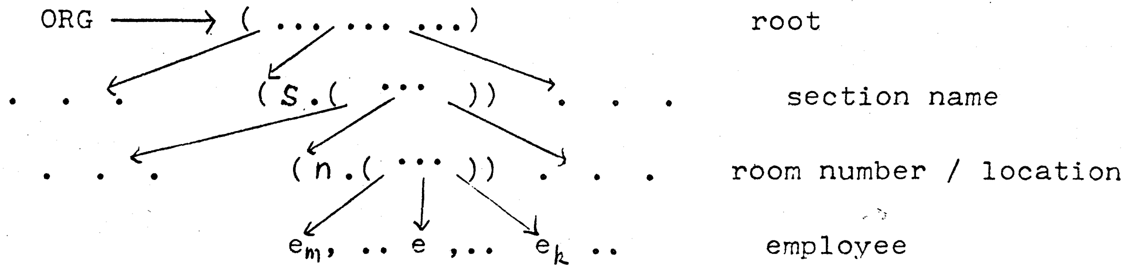
§7 Internal Representations of Relations and Domains

Physical representations for the relation exemplified in §2 and §6 are simple list structures of depth one, which can be thought as a sequential file. In this section we shall consider more complicated representations and show the versatility of our logical language in expressing various physical representations. Furthermore, the domains of variables will be discussed.

Suppose two relations, ROOM(Sect, Num) and EMP(Loc, Name) are defined at the conceptual level in a data base. If the location of each employee is identified with a room number in the ROOM relation, the two relations do not have to be represented independently. The two relations can be put together and implemented as a single tree structure

+) Using the LISP label operator, we may express this as
pr(Q(a)) = label[f; $\mathcal{P}$(f[pr(t)])].

which may be viewed as a hierarchical file. (See the figure below.)



ORG is the root of the tree structure. The implementations of ROOM and EMP by this structure are expressed as:

$$\mathrm{ROOM}(s,n) \equiv \exists x \exists y (x \leftarrow \mathrm{ORG} \wedge \mathrm{CAR}(x)=s \wedge y \leftarrow \mathrm{CDR}(x) \wedge \mathrm{CAR}(y)=n)$$

$$\mathrm{EMP}(1,e) \equiv \exists x \exists y (x \leftarrow \mathrm{ORG} \wedge y \leftarrow \mathrm{CDR}(x) \wedge \mathrm{CAR}(y)=1 \wedge e \leftarrow \mathrm{CDR}(y))$$

An implementation of a derived relation EMPSECT(e,s) defined as

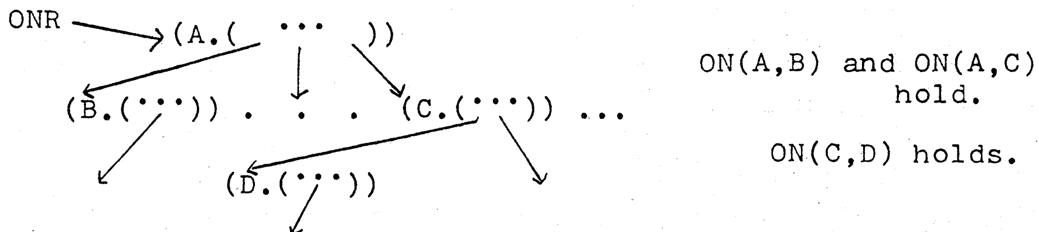
$$\mathrm{EMPSECT}(e,s) \equiv \exists z (\mathrm{ROOM}(s,z) \wedge \mathrm{EMP}(z,e))$$

can be expressed by

$$\mathrm{EMPSECT}(e,s) \equiv \exists x \exists y (x \leftarrow \mathrm{ORG} \wedge \mathrm{CAR}(x)=s \wedge y \leftarrow \mathrm{CDR}(x) \wedge e \leftarrow \mathrm{CDR}(y)).$$

Using this physical representation definition, an efficient access program for a query $Q = \{e \mid \mathrm{EMPSECT}(e, \mathrm{INFO})\}$ can be synthesized. The actual derivation of the program is given in [Y80].

The ON relation in §6 can be implemented as a tree structure in which a block represented by a parent node is on the block(s) represented by its child node(s). (See the figure below, ONR is the root.)



This physical representation of ON is expressed as:

$$\mathrm{ON}(a,b) \equiv \exists x (\mathrm{REACHABLE}(\mathrm{ONR},x) \wedge \mathrm{CAR}(x)=a \wedge \exists y (y \leftarrow \mathrm{CDR}(x) \wedge \mathrm{CAR}(y)=b)) \qquad (X)$$

where $\mathrm{REACHABLE}(x,y) \equiv y \leftarrow x \vee \exists z (z \leftarrow x \wedge \mathrm{REACHABLE}(\mathrm{CDR}(z),y))$

When the ON relation is implemented as above, the physical representation of the ABOVE relation should be expressed as:

$$ABOVE(a,b) \equiv \exists x(REACHABLE(ONR,x) \wedge CAR(x)=a \wedge \exists y(REACHABLE(CDR(x),y) \wedge CAR(y)=b)) \quad (XX)$$

But we lack a general theory(*) to prove the equivalence between the right-hand side of (XX) and the following formula which is obtained by substituting (X) for the occurrences of ON in the definition of ABOVE.

$$\exists x(REACHABLE(ONR,x) \wedge CAR(x)=a \wedge \exists y(y \leftarrow CDR(x) \wedge CAR(y)=b)) \vee \exists z(\exists x(REACHABLE(ONR,x) \wedge CAR(x)=a \wedge \exists y(y \leftarrow CDR(x) \wedge CAR(y)=z)) \wedge ABOVE(z,b)) \quad (XXX)$$

Fortunately we do not need such a theory to synthesize a program for the query $\{e \mid ABOVE(A,e)\}$ under this representation, if we apply the codification rule for recursive predicates to (XXX). An actual synthesis of the program is given in [Y80].

Notations of the form, domain(P.n), are used in logical transformation rules to express the domain of variable (or "attribute" in the data base terminology) in the n-th place of a predicate (or relation) P. When we use domains of variables in the process of program synthesis, actual enumerations of domain elements must be somehow provided. We assume that a list of domain elements can always be prepared for each variable used in relations which are defined at the conceptual level of the data base.(**) Such a list is introduced as a constant list into formulas in our logical language. The following example illustrates the use of such a list as well as the treatment of universal quantifiers in our method.

$$Q = \{r \mid \forall s ROOM(s,r)\}$$

This query means " list up rooms which are common to all sections". This is transformed by the rule (AS3).

$$\Rightarrow \{r \mid r \in domain(ROOM.2) \wedge \bigwedge_{\alpha \in domain(ROOM.1)} ROOM(\alpha,r)\}$$

(*) The work of Popplestone [P79] addresses some aspect of this problem.
(**) This assumption corresponds to the domain closure axiom in [R78].

The physical representations of ROOM is assumed to be the one defined in §2 and the domain(ROOM.2) and domain(ROOM.1) are assumed as constant lists RNUM (room number list) and SNAM (section name list), respectively. Thus,

$$\Rightarrow \{ r \mid r \leftarrow RNUM \wedge \bigwedge_{\alpha \leftarrow SNAM} \exists x (x \leftarrow ROOML \wedge CAR(x) = \alpha \wedge CDR(x) = r) \}$$

Using the rule(EL1a) and several other codification rules (e.g., CRS1, CRL2, CRL1), a program for the query is synthesized as

```
=>  generate[RNUM;
             id;
             λx.all-meet?[SNAM;
                          λα.some-meet?[ROOML;
                                        λβ.and[equal[car[β]; α]
                                               equal[cdr[β]; x]]]]]
```

§8 Discussions

<u>8.1 Optimization</u> Though the efficiency of a synthesized program depends mainly upon the simplicity of logical formulas before the application of codification rules, optimization of synthesized programs is often useful. For example, when a synthesized program contains a certain nested use of "generate" programs (i.e., an invocation of "generate" appears as an actual argument of another "generate" program), the nesting can be eliminated. Furthermore, combinations of "generate" and other programs (such as "all-meet?", "some-meet?" etc.) can be optimized. Typical cases of these optimizable combinations are given below.

```
(POR1)  generate[generate[l; λx.c[x] ; λx.p[x]] ;
                 λy.s[y] ;
                 λy.q[y]]
            =>  generate[l; λx.s[c[x]] ; λx.and[p[x]; q[c[x]]]]

(POR2)  some-meet?[generate[l; λy.c[y] ; λy.q[y]] ; λx.p[x]]
            =>  some-meet?[l; λx.and[p[c[x]]; q[x]]]

(POR3)  all-meet?[generate[l; λy.c[y] ; λy.q[y]] ; λx.p[x]]
            =>  all-meet?[l;  λx.or[not[q[x]]; p[c[x]]]]
```

It should be mentioned that Burstall and Darlington's optimization techniques [BD77] for recursive programs are also applicable.

8.2 Heuristics To synthesize an efficient program without "detour", we sometimes need heuristics guiding us when and which rules we should employ. We have the following rather general heuristics.

1. The rules whose left-hand side has an explicit membership predicate $\propto$ (e.g., AS1, ES1a) should have the higher priority.
2. The rules whose right-hand side has a domain notation (e.g., AS2, AS3) should have the lower priority.
3. In handling a set-type query, the rules which are applied to a set notation should have priority over the rules which are applied only to formulas. For example, the rules AS1, AS2 and AS3 have priority over the rules AL1 and AL2.

Beside these heuristics, we can use a set of heuristics which are based on Reiter's method [R78] for relational algebraic interpretations of queries that are expressed in relational calculus. (For more discussion, see [Y80] .)

8.3 Completeness and Termination

Our method for program synthesis is "complete" in the sense that for any query written in our logical language, we can always synthesize a program for the query. (See the remark at the end of §3.) Our method is also "complete" in the sense that a synthesized program returns all and the only answers to a given query if the synthesized program terminates. As noted at the end of §6, a synthesized program "above[a] " does not terminate if ON(x, x) or ABOVE(x, x) hold for some x.

8.4 Further Work

To construct a firm theoretical foundation for our approach is a good research topic. For example, it is interesting to establish the correctness of our logical and codification rules in some theoretical framework. Another example might be to develop a general theory in which we can show the equivalence of two predicates at least one of which is recursively defined (See §7 for an example).

Finally, we should not forget to remark that mechanization of our method is an interesting and promising project.

Acknowledgements

The author greatly benefited by several intensive discussions with K. Furukawa at ETL who gave him the seed of this research. The comments of H. Kakuda and his careful proof-reading were valuable.

References

[AN75] ANSI/X3/SPARC Study Group on Data Base Management System, Interim Report. Bulletin of ACM SIGMOD Vol.7, No.2 (1975)

[BD77] Burstall,R.D., and Darlington, J.,"A Transformation System for Developing Recursive Programs" JACM Vo.24,No.1 (1977)

[C78] Chang,C.L.,"DEDUCE 2: Further Investigations of Deduction in Relational Data Bases" in Logic and Data Bases Plenum Press (1978)

[CT77] Clark,K.L. and Tärnlund,S.A.,"A First Order Theory of Data and Programs" Proc. IFIP-77 (1977)

[D77] Date,C.J.,"An Introduction to Database Systems" 2nd Edition Addison Wesley (1977)

[G69] Green,C.,"Applications of Theorem Proving to Problem Solving" Proc. IJCAI-69 (1969)

[GMN78] Gallaire,H., Minker,J. and Nicolas,J.M., "An Overview and Introduction to Logic and Data Bases" in Logic and Data Bases Plenum Press (1978)

[HA79] Hanson,A. and Tärnlund,S.A., "A Natural Programming Calculus" Proc. IJCAI-79 (1979)

[M65] McCarthy,J. et al., "LISP 1.5 Programmer's Manual" MIT Press (1965)

[MW79] Manna,Z. and Waldinger,R.J., "A Deductive Approach to Program Synthesis" Proc.IJCAI-79 (1979)

[P79] Popplestone,R.J., "Relational Programming" Machine Intelligence Vol.9 (1979)

[R78] Reiter,R., "Deductive Question-answering on Relational Data Bases" in Logic and Data Bases Plenum Press (1978)

[S75] Sato,M., "Towards a Mathematical Theory of Programming Synthesis" Proc. IJCAI-79 (1979)

[WL69] Waldinger,R.J. and Lee,R.C., "PROW: A Step toward Automatic Programming Writing" Proc. IJCAI-69 (1969)

[Y80] Yonezawa,A. "A Non-theorem Proving Approach to Synthesis of Data Base Access programs" in preparation.

Appendix Other Logical Transformation Rules

(SE0) $\{s(x) \mid B(x)\} \Rightarrow \{s(\alpha) \mid B(\alpha) \wedge \alpha \in \mathrm{domain}(B.1)\}$

(SE1) $\{s(x) \mid P(x) \wedge Q(x)\} \Rightarrow \{s(\alpha) \mid P(\alpha) \wedge \alpha \in \{x \mid Q(x)\}\}$

(--a) $\{s(x) \mid P(x) \wedge Q(x)\} \Rightarrow \{s(x) \mid P(x)\} \cap \{s(x) \mid Q(x)\}$

(SE2) $\{s(x) \mid P(x) \wedge Q(x)\} \Rightarrow \{s(x) \mid P(x)\} \cup \{s(x) \mid Q(x)\}$

(SE3) $\{s(x) \mid R(x,t) \wedge t \propto L\} \Rightarrow \bigcup_{t \propto L} \{s(x) \mid R(x,t)\}$

(SE4) $\{s(x) \mid \bigvee_{t \propto L} R(x,t)\} \Rightarrow \bigcup_{t \propto L} \{s(x) \mid R(x,t)\}$

(SE5) $\{s(x) \mid \bigwedge_{t \propto L} R(x,t)\} \Rightarrow \bigcap_{t \propto L} \{s(x) \mid R(x,t)\}$

(SE6) $\{s(x) \mid \neg B(x)\} \Rightarrow \{s(\alpha) \mid \alpha \in \mathrm{domain}(B.1) - \{x \mid B(x)\}\}$

(SE7) $\{s(x) \mid x=c(y) \wedge P(y)\} \Rightarrow \{s(c(y)) \mid P(y)\}$

(--a) $\{s(x) \mid x=c(y)\} \Rightarrow \{s(c(y))\}$

(SE8) $\{x \mid x \in S\} \Rightarrow S$

(--a) $\{s(x) \mid x \in \{y \mid P(y)\}\} \Rightarrow \{s(x) \mid P(x)\}$

(SE9) $\{s(x) \mid R(x,t) \wedge P(t) \wedge c(t) \in \{c(y) \mid Q(y)\}\} \Rightarrow$
$\{s(x) \mid R(x,t) \wedge t \in \{y \mid Q(y)\}\}$ if Q(x) implies P(x).

(SE10) $\{s(x) \mid R(x,t) \wedge P(t) \wedge c(t) \in \{c(z) \mid Q(z)\}\} \Rightarrow$
$\{s(x) \mid R(x,t) \wedge Q(t)\}$ if Q(x) implies P(x).

Department of Information Scinences
Tokyo Institute of Technology
Oh-okayama, Meguro-ku, Tokyo, Japan

Research Reports on Information Sciences
Series C: Computer Science

C - 1. "A Note on Extending Equivalence Theories of Algorithms,"Kojiro Kobayashi, February 1974.

C - 2. "Generalizations of Regular Sets and thier Application to a Study of Context-Free Languages," Masako Takahashi, May 1974.

C - 3. "The Firing Squad Symchronization Problem for Arbitrary Two-Dimensional Arrays," Kojiro Kobayashi, November 1974.

C - 4. "Generalized Parenthesis Grammars and a Description of ALGOL," Masako Takahashi, February 1975.

C - 5. "Minimum Firing Time of the Two-Dimensional Firing Squad Synchronization Problem," Kojiro Kobayashi, May 1975.

C - 6. "A Report on a Symposium on Structured Programming and Experiences with it," Izumi Kimura, March 1976.

C - 7. "A Hashing Method for Fast Set Operations," Masataka Sassa & Eiichi Goto, June 1976.

C - 8. "A Minimal Time Solution to the Firing Squad Synchronization Problem of Rings with One-Way Information Flow," Kojiro Kobayashi, September 1976.

C - 9. "V-Tape, a Virtual Memory Oriented Data Type, and its Resource Requirements," Masataka Sassa & Eiichi Goto, January 1977.

C-10. "Rational Relations of Binary Trees," Masako Takahashi, March 1977.

C-11. "The Firing Squad Synchronization Problem for a Class of Polyautomata Networks," Kojiro Kabayashi, April 1977.

C-12. "On Positive Rational Relations of Binary Trees," Masako Takahashi, May 1977.

C-13. "On Teaching the Art of Compromising in the Development of External Specifications," Izumi Kimura, June 1977.

C-14. "On Proofreader's Programming," Izumi Kimura, August 1977.

C-15. "On Minimal Firing Time of Firing Squad Synchronization Problem for Polyautomata Networks," Kojiro Kobayashi, September 1977.

C-16. "Eliminating Unit Reductions from LR(k) Parsers Using Minimum Contexts," Takehiro Tokuda, April 1978.

C-17. "A Specification Technique for Abstract Data Types with Parallelism," Akinori Yonezawa, April 1978.

C-18. "Modelling Distributed Systems," Aki Yonezawa & Carl Hewitt, September 1978.

C-19. "A Pattern Matching Macro Processor," Masataka Sassa, October 1978

C-20. "Specifying and Verifying Software Systems with High Internal Concurrency Based on Actor Formalism," Akinori Yonezawa, January 1979.

C-21. "A Historical, Generalistic, and Complementary Approach in Introductory Computer Science Education," Izumi Kimura, January 1979.

C-22. "An Axion System for Rational Sets with Multiplicity," Masato Morisaki & Ko Sakai, January 1979.

C-23. "On Multitape Automata," Hideki Yamasaki, February 1979.

Department of Information Sciences
Tokyo Institute of Technology
Oh-okayama, Meguro-ku, Tokyo, Japan

Research Reports on Information Sciences
<u>Series C: Computer Science</u>

C-24. "Design and Implementation a Multipass-Compiler Generator," Masataka Sassa, Junko Tokuda, Tsuyoshi Shinogi & Kenzo Inoue, April 1979.

C-25. "A Derivation of Cook's Simulation Algorithm by Program Transformation," Osaum Watanabe, October 1979.

C-26. "A Note on Continuous Lattices," Kyoji Yuyama, December 1979.

C-27. "One Method for Constructing Self-Reproducing Programs," Kojiro Kobayashi, January 1980.

C-28. "A Double-Layered Text Editor," Hiroyasu Kakuda & Takashi Tsuji, March 1980.

C-29. "Eliminating Unit Reductions from LR(k) Parsers Using Minimum Contexts = A Revised Version of the Report C-16(April 1978)," Takehiro Tokuda, May 1980.