A Method for Synthesizing Data Retrieving Programs

AKINORI YONEZAWA*

A method is presented for synthesizing programs which retrieve stored data. Program specifications are given in the form of queries to a relational database. Queries are expressed in a language based on predicate calculus. Given a query at the conceptual level together with a description of physical representation of relations (or predicates), a program is synthesized which manipulates stored data at the physical 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 (or formulas) are successfully handled by our method. A target language (in which synthesized programs are written) is required to support functional arguments. Some aspects of future programming styles are discussed at the end of the paper.

1. Introduction

Program synthesis is the systematic derivation of a program from a specification. Most research activities in program synthesis rely on theorem proving techniques. In this approach, a program specification is stated as a theorem in some theory and proving the theorem is first attempted. If a successful proof for the theorem is constructed manually or mechanically, 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: (e.g., /G69/, and /WL69/ which are based on the resolution-type proof procedures and /HT79/ based on Gentzen's Natural Deduction System and /S79/based on the Goedel interpretation.) Another approach taken by Burstall and Darlington/BD78/ and Manna and Waldinger/MW79/ is based on the direct application of transformation or rewriting rules to the program specification. The recent work by Manna and Waldinger /MW80/combines induction and transformation rules within a single deductive system.

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 also uses theorem proving techniques. This paper presents a new method for synthesis of data retrieving programs, and also propose 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 procedural interpretations of logical formulas in which queries are stated. Given a query and the description of physical representations of relations (, both of which are stated in a first order language), a program is synthesized which actually manipulates physically stored data. Although no implementation exists presently, our approach is machine-oriented and the design of implementation is being undertaken. Fairly complex programs manipulating tree structures have been synthesized by hand simulation /Y81/.

*Dept. of Information Science, Tokyo Institute of Technology, Oh-okayama, Meguro-ku, Tokyo, Japan, 152.

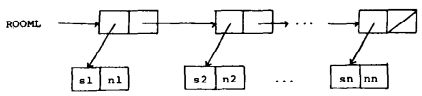
2. A Simple Example

To illustrate our program synthesis method, we 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. (See Figure below.)



This can be expressed as:

$$\mathrm{ROOM}(s, n) \equiv$$
$$\exists x(\mathrm{HEAD}(x)=s \wedge \mathrm{TAIL}(x)=n \wedge x \ll \mathrm{ROOML}),$$

where ROOML is a constant list, s and n are free

variables. In our language, $e \leqslant l$ is an abbreviation of the 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/.) HEAD and TAIL are one-place functions which give the first part of and the second part of a pair, respectively. We call this kind of logical equivalence which expresses the internal representation of a relation, a physical representation definition (PRD). Substituting this PRD, the query is transformed into:

$$\{ n \mid \exists x(\mathrm{HEAD}(x)=\mathrm{INFO} \wedge \mathrm{TAIL}(x)=n \wedge x \leqslant \mathrm{ROOML})\} \qquad (2\text{–}1)$$

This is logically equivalent to:

$$\{ n \mid \bigvee_{i\mathrm{R} \leqslant \mathrm{OOML}} (\mathrm{HEAD}(i)=\mathrm{INFO} \wedge \mathrm{TAIL}(i)=n)\}$$

where i is a new variable. By equality substitution we get:

$$\{ \mathrm{TAIL}(i) \mid \bigvee_{i \leqslant \mathrm{ROOML}} (\mathrm{HEAD}(i)=\mathrm{INFO})\}$$

Furthermore, this set is equivalent to:

$$\bigcup_{i \leqslant \mathrm{ROOML}} \{ \mathrm{TAIL}(i) \mid \mathrm{HEAD}(i)=\mathrm{INFO}\} \qquad (2\text{–}2)$$

The set expressed above is a collection of the second part of i which is an element of the list ROOML and whose first part is INFO. This suggests a procedural interpretation of the set notation (2–1), 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 i←car(L);
      if HEAD(i)=INFO then S←S ∪ {TAIL(i)};
        L←cdr(L)
  end;
return(S)                                   (P–1)
```

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 reduced and generalized by introducing a few types of transformation rules. By noting that the existentially quantified variable x in (2–1) behaves as an index which ranges over the list elements of ROOML, we adopt an E-quantifier elimination rule of the following form.

(ES3) $\{ t \mid \exists y(P(y) \wedge t=s(y))\} \Rightarrow \{s(\alpha) \mid P(\alpha)\}$

where α is a newly generated variable, $s(y)$ is a term constructed from variable y. Applying this rule to (2–1) with $P(y)$ being $\mathrm{HEAD}(x)=\mathrm{INFO} \wedge x \leqslant \mathrm{ROOML}$ and $s(y)$ being $\mathrm{TAIL}(x)$, we obtain:

$$\{ \mathrm{TAIL}(i) \mid \mathrm{HEAD}(i)=\mathrm{INFO} \wedge i \leqslant \mathrm{ROOML}\} \qquad (2\text{–}3)$$

We can (procedurally) interpret this notation in the same way as (2–2) and obtain the program (P–1).

To generalize the coding process, we introduce an operator pr which transforms a formula or set notation into a program that gives a procedural interpretation of (2–2). The above coding process (from (2–2) or (2–3) to (P–1)) can be viewed as an application of the following rule.

(CRS2) $\mathbf{pr}(\{s(x) \mid P(x) \wedge x \propto L\}) \ast\Rightarrow$
generate[**pr**(L); $\lambda x.$**pr**($P(x)$); $\lambda x.$**pr**($s(x)$)]

Note that L is a set or list and that $\propto$ stands for the set or list membership predicate. Generate $[l; p; s]$ is a procedure which takes a value argument l and two functional arguments p and s. This procedure returns a list of distinct items each of which is obtained by applying a function s to each element (in list l) which satisfies a condition (predicate) p. We assume that ignoring the order of elements, the returned list represents the set specified by $\{s(x) \mid \cdots\}$. $\lambda x.\langle \mathrm{body} \rangle$ is a notation for a procedure definition declaring x as a formal argument and $\langle$body$\rangle$ as the procedure body. Thus the right-hand side of (CRS2) 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 $\lambda x.$ **pr**($s(x)$) whose formal parameter is x and the body **pr**($s(x)$), and
—a procedure $\lambda x.$ **pr**($P(x)$) whose formal parameter is x and the body, **pr** ($P(x)$).

Applying the rule (CRS2) to (2–3) by instantiating $P(x)$ with $\mathrm{HEAD}(i)=\mathrm{INFO}$, L with ROOML and $s(x)$ with $\mathrm{TAIL}(i)$ and also using other simple coding rules, the following program (written in the fashion of LISP meta-expression/M65/) is obtained.

generate[ROOML; $\lambda x.$equal[car[x]; INFO]; $\lambda x.$cdr[x]]

Note that cdr, car, equal etc. denote procedures (functions) defined in LISP. We deliberately leave the definition of "generate" unspecified /Notel/* because many implementations are possible in many programming languages as long as they allow procedures as parameters of procedure invocation and support list or set structures. Note that if the target language does not support set structures, list structures are viewed as sets.

*/Notel/ An example of the body of "generate" in LISP is:

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

where nd[l] eliminates repetitive elements in list l.

3. Logical Language and Query Language

As illustrated in the previous section, both data base 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 manner and make its scope clear, we first discuss our logical language.

Roughly speaking, the language is an extension of first order predicate calculus augmented with equality, set membership and list membership. Terms, formulas, sets and queries are defined as follows.

(Terms)

1. An individual constant (written in capital letters) or variables (written in lower-case letters) are terms.
2. If f is an n-place function symbol (written in upper-case letters) and $t_1, \cdots, tn$ are terms, then $f(t_1, \cdots, tn)$ is a term.
3. All terms are constructed by 1 or 2.

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

(Formulas)

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 upper-case letters) and $t_1, \cdots, tn$ are terms, the $P(t1, \cdots, tn)$ is a formula.
4. If A and B are formulas, then $\neg A$, $A \vee B$, $A \wedge B$ and $A \rightarrow B$ are formula.
5. If A is a formula, x is a variable and l is a set or list, then $\forall xA$, $\exists xA$, $\bigvee_{x \propto l} A$ and $\bigwedge_{x \propto l} A$ are formulas.
6. All formulas are constructed through 1 to 5.

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 $t1, \cdots, tn$ are terms, $\{t1, \cdots, tn\}$ is a set whose elements are all the distinct members among $t1, \cdots, tn$.
3. If $t(x)$ is a term containing a free variable x and $P(x)$ is a formula with free occurrences of x, then $\{t(x) | P(x)\}$ is a set whose elements are distinct terms $t(x)$ such that x satisfies $P(x)$. $t(x)$ and $P(x)$ may contain other free variables, but $t(x)$ may not contain bound variables. When $t(x)$ is a variable x itself, 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 s t 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 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 $\{t(x) | P(x)\}$ 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 formulas defined above and requests the system to return the truth value (yes or no) of a 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 universal theorem proving. But our synthesis method restrict itself to queries which can be answered by searching 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 coding rules. We shall discuss the former ones in this section.

As the example in the second section suggests, general algorithmic structures of synthesized programs are summarized 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 condition, do something on the item or report the truth value of the conditions." This general algorithm is usually used in a nested manner. The list ROOML in the example in the second section is such a source of data items. Successful synthesis of efficient programs depends upon the discovery and choice of data item sources which may be implicit in formulas.

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

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

where $s(x)$ is a term containing a free variable x 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.)

$$\{s(x) | \exists y(R(x, y) \wedge y \propto L)\} \Rightarrow \{s(x) | R(x, \alpha) \wedge \alpha \propto L\} \qquad (*)$$

$$\exists y(P(y) \wedge y \propto L) \Rightarrow \bigvee_{\alpha \propto L} P(\alpha) \qquad (**)$$

L in the predicates (of the form) $x \propto L$ provides the source of data items in these rules. If no such forms of predicates are found in formulas, for example,

$$\{s(x) | \exists y(R(x, y) \wedge Q(y))\}, \exists y(P(y) \wedge Q(y)),$$

we need to create such predicates. The following rules do this.

(ES1) $\{s(x)|\exists y(R(x, y) \wedge Q(y))\} \Rightarrow \{s(x)|R(x, \alpha) \wedge \alpha \in \{y|Q(y)\}\}$

(EL1) $\exists y(P(y) \wedge Q(y)) \Rightarrow \bigvee_{\alpha \in \{y|Q(y)\}} P(\alpha)$

An explicit set (or list) L in the previous rules (*) and (**) are replaced by one specified by a predicate Q. This idea is based on more general logical equivalence.

$$P(x) \wedge Q(x) \Leftrightarrow P(x) \wedge x \in \{t|Q(t)\}$$

Sometimes the sources are hard to find; for example,

$$\{s(x)|\exists y A(x, y)\}, \quad \exists y B(y)$$

where $A(x, y)$ and $B(y)$ cannot break into the patterns of (ES1) or (EL1). In such cases, we must introduce the domains of quantified variables. This is expressed as:

(ES2) $\{s(x)|\exists y A(x, y)\} \Rightarrow \{s(x)|A(x, \alpha) \wedge \alpha \in \mathrm{dom}(A.2)\}$

(EL2) $\exists y B(y) \Rightarrow \bigvee_{\alpha \in \mathrm{dom}(B.1)} B(\alpha)$

dom($P.n$) denotes the domain of a variable in the n-th place of a predicate (or relation) P.

For formulas containing universal quantifiers, similar rules are adopted, the motivation being the same. The list of transformation rules for quantifiers is shown in Fig. 1. Rules for other logical symbols and simplification rules for set notations are given in the Appendix. Besides these rules, we use the rules for replacement of logically equivalent formulas, rules for renaming of bound variables, and rules for equality substitution and rules for definition substitution. Furthermore, conventional logical rules (such as $A \wedge B \Rightarrow B \wedge A$) can also be used.

—Universal Quantifier—

(UL1)* $\forall y(Q(y) \rightarrow P(y)) \Rightarrow \bigwedge_{\alpha \in \{y|Q(y)\}} P(\alpha)$

(UL2) $\forall y P(y) \Rightarrow \bigwedge_{\alpha \in \mathrm{dom}(P.1)} P(\alpha)$

(US1)* $\{s(x)|\forall y(Q(y) \rightarrow R(x, y))\} \Rightarrow \{s(x)|(\bigwedge_{\alpha \in \{y|Q(y)\}} R(x, \alpha)) \wedge x \in \mathrm{dom}(R.1)\}$

(US2) $\{s(x)|\forall y R(x, y)\} \Rightarrow \{s(x)|(\bigwedge_{\alpha \in \mathrm{dom}(R.2)} R(x, \alpha)) \wedge x \in \mathrm{dom}(R.1)\}$

—Existential Quantifier—

(EL1)* $\exists y(P(y) \wedge Q(y)) \Rightarrow \bigwedge_{\alpha \in \{y|Q(y)\}} P(\alpha)$

(EL2) $\exists y B(y) \Rightarrow \bigvee_{\alpha \in \mathrm{dom}(B.1)} B(\alpha)$

(EL3) $\exists y(s(t) = y \wedge P(y)) \Rightarrow P(s(t))$

(ES1)* $\{s(x)|\exists y(R(x, y) \wedge Q(y))\} \Rightarrow \{s(x)|R(x, \alpha) \wedge \alpha \in \{y|Q(y)\}\}$

(ES2) $\{s(x)|\exists y A(x, y)\} \Rightarrow \{s(x)|A(x, \alpha) \wedge \alpha \in \mathrm{dom}(A.2)\}$

(ES3) $\{t|\exists y(t = s(y) \wedge P(y))\} \Rightarrow \{s(y)|P(y)\}$

*When $Q(y)$ in UL1, US1, EL1 and ES1 is of the form $y \propto L$, $\alpha \in \{y|Q(y)\}$, in the right hand side of each rule, it is simplified into $y \propto L$.
(Note: $t \in \{x|x \propto L\} \leftrightarrow t \propto L$)

Fig. 1 Transformation Rules for Quantifiers.

5. Coding Rules

To obtain a procedural interpretation for a set notation or formula (namely, to translate such a notation into a program), we use the coding rules listed in Fig. 2. Most rules are expressed by recursive applications of the coding operator **pr**. An application of **pr** to 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/M65/, actual implementations of the procedures may be written in other languages as long as they have the functional argument facility and support list processing primitives.

The informal meaning of each procedure appeared in the coding rules 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; p; s]: list items (without repetition) each of which is obtained by applying a function s to each element of l which satisfies a predicate p.
- some [l; p]: ask whether or not some element of a list l satisfies a predicate p.
- every [l; p]: ask whether or not all the elements of a list l satisfy a predicate p.
- i-union [l; f]: return a list whose elements are union of sets each of which is obtained by applying a function f to each element of a list l.
- i-intersect [l; f]: return a list whose elements are the intersection of sets each of which is obtained by applying a function f to each element of a list l.

6. Recursive Relations and Recursive Programs

To accommodate a wide variety of queries, it is often necessary to use relations which are not stored explicitly in the data base, but which are logically derivable from explicitly stored relations.

When a relation is defined solely in terms of explicitly stored relations, no complication arises and we can simply substitute the derived relation by its definition. But consideration is needed when the definition contains itself, namely the relation is defined recursively. An example of such relations 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

—Set—

(CRS1) $\mathbf{pr}(\{x|x \propto L\}) \Rightarrow \mathbf{pr}(L)$

(CRS2) $\mathbf{pr}(\{s(x)|P(x) \wedge x \propto L\}) \Rightarrow$ generate[$\mathbf{pr}(L)$; $\lambda x.\mathbf{pr}(P(x))$; $\lambda x.\mathbf{pr}(s(x))$]

(CRS3) $\mathbf{pr}(\{s(x)|R(x,t) \wedge t \propto L\}) \Rightarrow$ i-union[$\mathbf{pr}(L)$; $\lambda t.\mathbf{pr}(\{s(x)|R(x,t)\})$]

(CRS4) $\mathbf{pr}(\bigcap_{t \propto L} S(t)) \Rightarrow$ i-intersect[$\mathbf{pr}(L)$; $\lambda t.\ \mathbf{pr}(S(t))$]; $S(t)$ denotes a set with index t.

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

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

(CRS7) $\mathbf{pr}(S-T) \Rightarrow$ difference[$\mathbf{pr}(S)$;$\mathbf{pr}(T)$]

(CRS8) $\mathbf{pr}(\{t1, \cdots, tn\}) \Rightarrow$ list[$\mathbf{pr}(t1)$; $\cdots$; $\mathbf{pr}(tn)$]

(CRS9) $\mathbf{pr}(\{n|(P \wedge Q(n)) \wedge (\neg P \wedge R(n))\}) \Rightarrow$ [$\mathbf{pr}(P) \rightarrow \mathbf{pr}(\{n|Q(n)\})$; $\mathrm{T} \rightarrow \mathbf{pr}(\{n|R(n)\})$]

where a predicate P does not contain n. (conditional)

—Formula—

(CRL1) $\mathbf{pr}(s(t) \propto L) \Rightarrow$ some[$\mathbf{pr}(L)$; $\lambda x.$ equal[x; $\mathbf{pr}(s(t))$]]

(CRL2) $\mathbf{pr}(\bigvee_{t \propto L} P(t)) \Rightarrow$ some[$\mathbf{pr}(L)$; $\lambda t\mathbf{pr}.(P(t))$]

(CRL3) $\mathbf{pr}(\bigwedge_{t \propto L} P(t)) \Rightarrow$ every[$\mathbf{pr}(L)$; $\lambda t.\mathbf{pr}(P(t))$]

(CRL4) $\mathbf{pr}(\neg P) \Rightarrow$ not[$\mathbf{pr}(P)$],
$\mathbf{pr}(P \vee Q) \Rightarrow$ or[$\mathbf{pr}(P)$; $\mathbf{pr}(Q)$],
$\mathbf{pr}(P \wedge Q) \Rightarrow$ and[$\mathbf{pr}(P)$; $\mathbf{pr}(Q)$]

(CRL5) $\mathbf{pr}(t1 = t2) \Rightarrow$ equal[$\mathbf{pr}(t1)$; $\mathbf{pr}(t2)$]

(CRL6) $\mathbf{pr}(t = \mathrm{NIL}) \Rightarrow$ null[$\mathbf{pr}(t)$]

—Term—

(CRT1) $\mathbf{pr}(\mathrm{HEAD}(t)) \Rightarrow$ car[$\mathbf{pr}(t)$]

(CRT2) $\mathbf{pr}(\mathrm{TAIL}(t)) \Rightarrow$ cdr[$\mathbf{pr}(t)$]

(CRT3) $\mathbf{pr}(\mathrm{CONSTL}(t1, t2)) \Rightarrow$ cons[$\mathbf{pr}(t1)$; $\mathbf{pr}(t2)$]

(CRT4) $\mathbf{pr}(\mathrm{ID}(t)) \Rightarrow \mathbf{pr}(t)$

(CRT5) $\mathbf{pr}(\langle$constant-or-variable$\rangle) \Rightarrow \langle$constant-or-variable$\rangle$

Fig. 2 Coding Rules.

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 relation, we can issue a query

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

which requests to return a set of all the blocks below a specified 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 ($a \leftarrow A$, $b \leftarrow e$)).

$$\{e|\ \mathrm{ON}(A, e) \wedge \exists z(\mathrm{ON}(A, z) \wedge \mathrm{ABOVE}(z, e))\} \quad (6\text{–}1)$$

Our strategy to deal with the recursion is to leave the recursive occurrence of ABOVE(z, e) in the formula intact and substitute a physical representation definition of ON for the two occurrences 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:

$$\mathrm{ON}(a, b) \equiv \exists x(x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = a \wedge \mathrm{TAIL}(x) = b)$$

Substituting the right-hand side for the occurrences of ON in (6–1), we obtain:

$$\left\{ e \,\middle|\, \begin{array}{l} \exists x(x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A \wedge \mathrm{TAIL}(x) = e) \vee \\ \exists z(\exists x(x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A \wedge \mathrm{TAIL}(x) = z) \wedge \\ \qquad \mathrm{ABOVE}(z, e)) \end{array} \right\}$$

The existential quantifier on z can be eliminated by (EL3).

$$\left\{ e \,\middle|\, \begin{array}{l} \exists x(x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A \wedge \mathrm{TAIL}(x) = e) \vee \\ \exists x(x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A \wedge \\ \qquad \mathrm{ABOVE}(\mathrm{TAIL}(x), e)) \end{array} \right\}$$

Factoring common expressions by the distributive law, this is simplified into:

$$\{e|\exists x((x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A) \wedge (\mathrm{TAIL}(x) = e \vee \mathrm{ABOVE}(\mathrm{TAIL}(x), e)))\}$$

Now we can eliminate the existential quantifier on x by using (ES1).

$$\{e|(\mathrm{TAIL}(\alpha) = e \vee \mathrm{ABOVE}(\mathrm{TAIL}(\alpha), e)) \wedge \alpha \in \{x|x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A\}\}$$

Using a coding rule (CRS3), we obtain:

$$i\text{-union}[\mathbf{pr}(\{x|x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A\}); \lambda\alpha.\mathbf{pr}(\{e|\mathrm{TAIL}(\alpha) = e \vee \mathrm{ABOVE}(\mathrm{TAIL}(\alpha), e)\})]$$

The second parameter of i-union is transformed by (CRS5) and a set equivalence rule (SE2) in the Appendix.

$$i\text{-union}[\mathbf{pr}(\{x|x \prec \mathrm{ONL} \wedge \mathrm{HEAD}(x) = A\}); \lambda\alpha.\ \mathrm{union}[\mathbf{pr}(\{e|\mathrm{TAIL}(\alpha) = e\}); \mathbf{pr}(\{e|\mathrm{ABOVE}(\mathrm{TAIL}(\alpha), e)\})]]$$

Several applications of coding rules produce:

$$i\text{-union}[\mathrm{generate}[\mathrm{ONL}; \lambda x.\mathrm{equal}[\mathrm{car}[x]; A]; \lambda x.x]; \lambda\alpha.\mathrm{union}[\mathrm{list}[\mathrm{cdr}[\alpha]]; \mathbf{pr}(\{e|\mathrm{ABOVE}(\mathrm{TAIL}(\alpha), e)\})]] \quad (6\text{–}2)$$

by CRS2, CRL5, CRT1, CRT2, CRT5, CRS8.

Here we wish to eliminate the **pr** operator applied to

$\{e|\text{ABOVE}(\cdots)\}$. To do so, we need the following special coding rule which introduces recursive invocations into the program to be synthesized.

Let $Q(a)$ be a query containing a free variable a. When the result of coding of $Q(a)$ contains occurrences of $\mathbf{pr}(Q(t))$ where t is a term, namely,

> if $\mathbf{pr}(Q(a))$ becomes $\mathscr{P}(\mathbf{pr}(Q(t)))$,
> then $\mathbf{pr}(Q(a))=f[a]$ where $\lambda x.f[x]=\mathscr{P}(f[\text{pr}(t)])$

Applying this rule to (6–2), we obtain a program for the original query.

$$\mathbf{pr}(\{e|\text{ABOVE}(A, e)\})=p\text{-above}[A]$$

where

p-above$[z]=$

i-union[generate[ONL; λx.equal[car$[x]$;z];$\lambda x.x$];

$\lambda\alpha$. union[list[cdr$[\alpha]$]; p-above[cdr$[\alpha]$]]]

Note that the synthesized program p-above$[A]$ does not terminate if ON(x, x) or ABOVE(x, x) hold for some x.

Recursively defined relations sometimes produce programs which do not terminate for any input. For example, if the ABOVE relation is defined in the following way:

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

the program pp-above$[A]$ synthesized for the same query $\{e|\text{ABOVE}(A, e)\}$ does not terminate for any physical representation of the relation ON. The following derivation explains this phenomenon.

$\{e|\text{ABOVE}(A, e)\}\Rightarrow$

$\{e|\text{ON}(A, e)\} \cup \{e|\exists z(\text{ABOVE}(A, z) \wedge \text{ABOVE}(z, e))\}$

(substitution of definition and (SE2))

$\Rightarrow\{e|\text{ON}(A, e)\} \cup \{e|\text{ABOVE}(\alpha, e) \wedge \alpha \in \{z|\text{ABOVE}(A, z)\}\}$ (ES1)

Applying coding rules and the special rule for recursion to this set notation (the same physical representation for ON is assumed), we obtain the program:

pp-above$[A]=$

union[generate[ONL; λx.equal[car$[x]$; A];λx.cdr$[x]$];

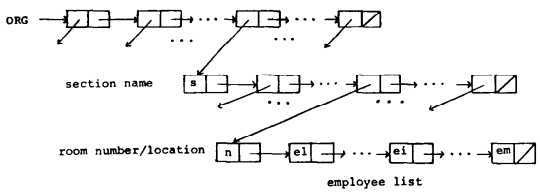
i-union[pp-above$[A]$; $\lambda\alpha.pp$-above$[\alpha]$].

Since an execution of pp-above$[A]$ always invoke pp-above$[A]$ itself as an actual argument of i-union, this program always runs forever. This leads to the fact that our coding process can detect some of the non-terminating programs (,though the general case is known to be undecidable). This fact corresponds that compilers for conventional programming languages would detect non-terminating programs (see the discussion in Sec. 8.5)

7. Physical Representation of Relations and Domains

Physical representations for the relations exemplified in Sec. 2 and Sec. 6 are simple list structures of depth one, which might be thought of as a sequential file. In this section we shall consider more complex 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 relation can be put together and implemented as a single tree structure, which may be viewed as a hierarchical files. (See the figure below)



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

$$\text{ROOM}(s, n)\equiv\exists x\exists y(x\lessdot\text{ORG} \wedge \text{HEAD}(x)=s \wedge y\lessdot\text{TAIL}(x) \wedge \text{HEAD}(y)=n)$$

$$\text{EMP}(l, e)\equiv\exists x\exists y(x\lessdot\text{ORG} \wedge y\lessdot\text{TAIL}(x) \wedge \text{HEAD}(y)=l \wedge e\lessdot\text{TAIL}(y))$$

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

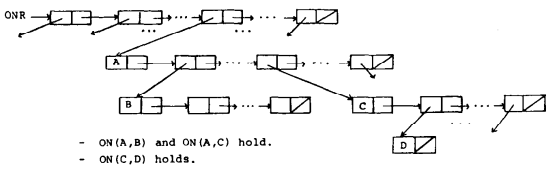
$$\text{EMPSECT}(e, s)\equiv\exists z(\text{ROOM}(s, z) \wedge \text{EMP}(z, e))$$

can be expressed by

$$\text{EMPSECT}(e, s)\equiv\exists x\exists y(x\lessdot\text{ORG} \wedge \text{HEAD}(x)=s \wedge y\lessdot\text{TAIL}(x) \wedge e\lessdot\text{TAIL}(y)).$$

Using this physical representation definition, an efficient access program for a query $Q=\{e|\text{EMPSECT}(e, \text{INFO})\}$ can be synthesized. The actual derivation of a program which manipulate more complex tree structures is found in /Y81/.

The ON relation in Sec. 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:

$$\text{ON}(a,b)\equiv\exists x(\text{REACHABLE}(\text{ONR}, x)\wedge\text{HEAD}(x)=a\wedge\exists y(y\lessdot\text{TAIL}(x) \wedge \text{HEAD}(y)=b)) \quad (7\text{–}1)$$

where

REACHABLE(x, y)
$\equiv y \prec x \vee \exists z(z \prec x \wedge \text{REACHABLE}(\text{TAIL}(z), y))$

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

$$\text{ABOVE}(a, b) \equiv \exists x(\text{REACHABLE}(\text{ONR}, x) \wedge \text{HEAD}(x) = a \wedge \exists y(\text{REACHABLE}(\text{TAIL}(x), y) \wedge \text{HEAD}(y) = b)) \quad (7\text{–}2)$$

If we used this definition to obtain the program for a query $\{e|\text{ABOVE}(A, e)\}$, a rather simple program would be synthesized. But we lack a general theory /Note 3/ to prove the equivalence between the right-hand side of (7–2) and the following formula which is obtained by substituting (7–1) for the occurrences of ON in the definition of ABOVE in Sec. 6.

$$\exists x(\text{REACHABLE}(\text{ONR}, x) \wedge \text{HEAD}(x) = a \wedge \exists y(y \prec \text{TAIL}(x) \wedge \text{HEAD}(y) = b)) \vee \exists z(\exists x(\text{REACHABLE}(\text{ONR}, x) \wedge \text{HEAD}(x) = a \wedge \exists y(y \prec \text{TAIL}(x) \wedge \text{HEAD}(y) = z)) \wedge \text{ABOVE}(z, b)) \quad (7\text{–}3)$$

Fortunately we do not need such a theory for the query, because we can synthesize a program for the query by directly applying the coding rule for recursive predicates to (7–3).

Notations of the form, dom ($P.n$), are used in logical transformation rules to express the domain of variables (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 /Note 4/. 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.

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

This query means "list up rooms which are common to all the sections." This is transformed by the rule (US2).

$$\{r|(\bigwedge_{\alpha \in \text{dom(ROOM.1)}} \text{ROOM}(\alpha, r)) \wedge r \in \text{dom(ROOM.2)}\}$$

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

$$\{r|(\bigwedge_{\alpha \in \text{SNAM}} \exists x(\text{HEAD}(x) = \alpha \wedge \text{TAIL}(x) = r \wedge x \prec \text{ROOML})) \wedge r \in \text{RNUM}\}$$

Using the rule (EL1) and several other coding rules (e.g., CRS2, CRL2, CRL3), a program for the query is synthesized as:

```
generate[RNUM;
        λx.every[SNAM;
                λα.some[ROOML;
                        λβ.and[equal[car[β]; α];
                               equal[cdr[β]; x]]]]
        λx.x]
```

/Note 3/ The work of Popplestone /P79/ addresses some aspect of this problem.

/Note 4/ This assumption corresponds to the domain closure axiom in /R78/.

8. Concluding Remarks

8.1 Optimization

Though the efficiency of a synthesized program depends mainly upon the simplicity of logical formulas before the application of coding 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 unfolded. Furthermore, combinations of "generate" and other procedures (such as "every," or "some") can be optimized. Typical cases of these optimizable combinations are given below.

(POR1) generate[generate[l; $\lambda x.p[x]$; $\lambda x.s[x]$] $\lambda y.q[y]$; $\lambda y.c[y]$] $\Rightarrow$ generate[l; λx.and[$p[x]$; $q[s[x]]$]; $\lambda x.c[s[x]]$]

(POR2) some[generate[l; $\lambda y.q[y]$; $\lambda y.s[y]$]; $\lambda x.p[x]$] $\Rightarrow$ some[l; λx.and[$p[s[x]]$; $q[x]$]]

(POR3) every[generate[l; $\lambda y.q[y]$; $\lambda y.s[y]$]; $\lambda x.p[x]$] $\Rightarrow$ every[l; λx.or[not[$q[x]$]; $p[s[x]]$]]

It should be mentioned that Burstall and Darlington's optimization techniques /BD77/ for recursive programs are also applicable. Furthermore, various optimization techniques are applicable once function calls are expanded into function bodies.

8.2 Target Languages

Our method does not restrict itself to LISP-like languages as target languages (in which synthesized programs are written). By extending the logical language to include appropriate functions and replacing the coding rules (mainly, rules for terms) with suitable ones, we can synthesize PASCAL-like programs which use record structures and pointers.

8.3 Completeness

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 Sec. 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.

8.4 Theoretical Foundation

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 coding 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 Sec. 7 for an example).

8.5 Higher Order Functions and Future Programming Style

The synthesis method presented in this paper extensively uses functional arguments in the phase of application of coding rules and thus resulting synthesized programs are structured through higher order functions. It is the author's belief that the use of higher order functions (i.e. programs which take functions or programs as parameters or result values) will gain increasing importance in the future programming style, which will be based on very high level programming languages.

By "very high" level languages, we mean logical languages or pure applicative (or functional) languages augmented with data structuring features; we do not mean natural languages, which we expect will face the hard barrier of semantic ambiguity before they will be used as effective specification languages. In this context our present work can be viewed as one suggesting a method of translating a logical language in a functional language: the latter is relatively lower than the former, though both are still high level languages in the current standard. Yet in some ten years, with the accumulation of knowledge about various translation (or transformation) rules the programming method suggested by our approach will be considered as "compilation" rather than "synthesis." This view will become more evident with the availability of machine architectures suitable for functional programming.

References

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

/BD77/ BURSTALL, R. D. and DARLINGTON, J. A Transformation System for Developing Recursive Programs, JACM **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 TARNLUND, S. A. A First Order Theory of Data and Programs, Proc. IFIP-77 (1977).

/D77/ DATE, C. J. *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 NICHOLAS, J. M. An Overview and Introduction to Logic and Data Bases, in *Logic and Data Base*, Plenum Press (1978).

/HT79/ HANSON, A. and TARNLUND, 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. Synthesis: Dreams⇒ Programs, IEEE. Trans, Softw. Eng. SE-5, No. 4 (1979).

/MW80/ MANNA, Z. and WALDINGER, R. A Deductive Approach to Program Synthesis, TOPLAS **2**, No. 1 (1980).

/P79/ Popplestone, R. J. Relational Programming, *Machine Intelligence* **9** (1979).

/R78/ REITER, R. Deductive Question-answering on Relational Data Bases, in *Logic and Data Bases*, Plenum Press (1978).

/S79/ 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 Program Writing, Proc. IJCAI-69 (1969).

/Y80/ YONEZAWA, A. A Method for Synthesis of Data Base Access Programs, Res. Rep. C-30, Dept. of Info. Sci., Tokyo Inst. of Tech. (1980).

/Y81/ Program Synthesis, Proc. of Summer Symposium on Programming. Info. Processing Society of Japan (1981) in Japanese.

Appendix

(SE1) $\{s(x)|B(x)\} \Rightarrow \{s(\alpha)|B(\alpha) \wedge \alpha \in \mathrm{dom}\,(B.1)\}$

(SE2) $\{s(x)|P(x) \vee Q(x)\} \Rightarrow \{s(x)|P(x)\} \cup \{s(x)|Q(x)\}$

(SE3) $\{s(x)|P(x) \wedge Q(x)\} \Rightarrow \{s(x)|P(x)\} \cap \{s(x)|Q(x)\}$

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

(SE5) $\{s(x)|\bigvee_{t \propto L} R(x,t)\} \Rightarrow \{s(x)|R(x,t) \wedge t \propto L\}$

(SE6) $\{s(x)\bigwedge_{t \propto L} R(x,t) \Rightarrow \bigcap_{t \propto L} \{s(x)|R(x,t)\}$

(SE7) $\{s(x)|\neg B(x)\} \Rightarrow \{s(\alpha)|\alpha \in \mathrm{dom}\,(B.1) - \{x|B(x)\}\}$

(SE8) $\{s(x)|x=c(y) \wedge P(y)\} \Rightarrow \{s(c(y))|P(y)\}$

(SE9) $\{x|x \in S\} \Rightarrow S$

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

(Received July 14, 1981: revised September 18, 1981)