FORMAL SPECIFICATION AND IMPLEMENTATION OF OPERATIONS IN INFORMATION MANAGEMENT SYSTEMS

Erik Sandewall
Software Systems Research Center
Linköping University
Linköping
Sweden

ABSTRACT

Among information management systems we include general purpose systems such as text editors and data editors (forms management systems) as well as special purpose systems such as mail systems and computer based calendars. Based on a method for formal specification of some aspects of IMS, namely the structure of the data base, the update operations and the user dialogue, this paper shows how reasonable procedures for executing IMS operations can be written in the notation of a first-order theory in such a way, that the procedure is a logical consequence of the specification.

1. SPECIFICATION OF IMS AND THE FORMAL IMPLEMENTATION PROBLEM

Information management systems (IMS) include general-purpose systems such as text editors and data editors (e.g. forms management systems), and special purpose systems such as mail systems and computer based calendars. An IMS provides an interactive service for 'moving data around' or 'general housekeeping': entering data into the computer; changing it; displaying part of the data through a 'window' on the screen while it is being entered or changed; rearranging it and printing it out on various media.

The computing world abounds with IMS: there are IMS for various kinds of information (text, structured data, graphical data, etc.) as well as for various applications. As Boehm has shown [BOE80] the implementation of an interactive application program consists to a large extent of implementing a number of IMS services. In the academic environment as well, every hacker writes his own editors: not just once, but often several times. Programming environments, which are presently an area of high research interest, are IMS for software.

In spite of this proliferation of IMS, there is a striking lack of systematization or formal understanding of what this kind of software really does. In fact, the lack of formal understanding is probably one reason for the proliferation: various IMS perform

This research was supported by the Swedish Board of Technical Development.

very similar tasks and it is reasonable to believe that there could be a design which is simpler and more powerful at the same time. A formal specification of IMS can serve this purpose if it is clear and easy to understand, and if it can accommodate the application specific details as well as the generalities that are present in all IMS.

In a previous paper [SAN82] we have described a method which allows us to *express in precise terms* the services which are provided by actual IMS (both those services which are characteristic for most IMS as well as a framework for characterizing application-specific services). In particular, the following things are characteristic for IMS (see [SAN82]):

- the existence of a *data repository* where information is stored;
- *update operations* on the stored data;
- the existence of a *focus of attention* (often represented as the cursor position) relative to which the edit operations are performed;
- *display operations* on the screen - involving a traversal of the structure at hand so that its parts can be displayed individually, and layout planning so that the whole display makes sense;
- dialogue interpretation, particularly the *common command loop*. An additional problem is the *prompting* situation where the user is supposed to provide an answer to a question or a specific piece of data (in data entry). However, even in prompting situations the user must be able to override the context by entering special commands (often implemented using control characters).

Our approach to IMS *separates* the specification of the contents of the display from the specification of the effects of operations on the data repository.

The topic of the present paper is to describe a method for transforming the specification of a set of operations (i.e. of their effect on the data repository) into a procedure which executes them. We use the term *implementation* for the transformation from a specification to a procedure which satisfies it. In the present paper, the resulting procedure is expressed in a simple language which we introduce here, but which has the characteristic properties of conventional programming languages: local variables, assignment statements (although using 'single assignment'), operations which 'update' the data repository, conditionals, recursion, etc. At the same time, the language for expressing procedures is chosen from a restricted first order theory, and the relation between the specification and the implementation is one of logical consequence. The paper contains specifications of the proposed languages, as well as some examples of how an operation can be implemented in the language.

2. NOTATION

In order to write specifications for the IMS operations, we must first define the domains for data structures in the date repository of the IMS. Following the advice of Blikle [BLI82], we shall express our domain equations in set theory notation. The notation will therefore be the standard notation of predicate logic and set theory, with only a small number of notational extensions which agree with Blikle and/or agree roughly with the practice of the denotational semantics literature. We use the following notation:

Predicate logic

$\wedge$ and

$\vee$ or

$\Rightarrow$ implies

$\equiv$ logical equivalence

$\exists$ existential quantifier

$\forall$ universal quantifier (free variables are viewed as universally quantified)

Set theory

$\cup$ union of sets

$\in$ membership in sets (particularly domains)

$\subset$ subset relation between sets

$<x,y,\ldots>$
The sequence whose members are x, y, etc. $<x>$ is distinct from x. $g.x$ is the result of applying the function g to the argument x. Also written $g[x]$. For functions of several arguments only the latter notation is used.

hd $hd.<x_1,x_2,\ldots,x_n>=x_1$

tl $tl.<x_1,x_2,\ldots,x_n>=<x_2,\ldots,x_n>$

pfx $pfx(x,<x_1,\ldots,x_n>)=<x,x_1,\ldots,x_n>$

conc concatenation of sequences:
$<x_1,\ldots,x_k>\ conc\ <x_{k+1},\ldots,x_n>=<x_1,\ldots,x_n>$.
This operation is extended to sets of sequences in the obvious way:
$A\ conc\ B=\{a\ conc\ b \mid a\in A\ \wedge\ b\in B\}$.

rev reversal of sequences

subst substitution in a sequence structure: $subst[old,new,x]$ replaces every occurrence of *old* by an occurrence of *new* in the structure x.

$\times$ Cartesian product: $A\times B\times\ldots\times D$ is defined as the set of all $<a,b,\ldots,d>$ where $a\in A$, etc.

! set of onetuples: $A!=\{<a> \mid a\in A\}$

* $A^*=\{<a_1,\ldots,a_n> \mid a_i\in A\ \wedge\ n>0\}$

$^+$ $A^+=\{<a_1,\ldots,a_n> \mid a_i\in A\ \wedge\ n>1\}$

$\rightsquigarrow$ pseudo-mapping: $A\rightsquigarrow B \mid e$ is the set of all total functions from A to B such that $f[a]\neq e$ for only a finite number of arguments. If f is a pseudomapping for which e is *nil* we shall not distinguish it from the set of all pairs $<x,f[x]>$ where $f[x]\neq nil$. In particular, no distinction will be made between the pseudomapping which maps everything to *nil* and the empty set.

Named domains and selector functions

The symbols introduced above are used for writing domain equations for a collection of *named domains*. Often a domain A is defined by an equation of the form $A=B\times C\times D\ldots$. The components of a member of A can then be selected using the functions *hd* and *tl* defined above. It is convenient to introduce the following, more mnemonic notation:

$s-$ $s-b$ is a function $A \rightarrow B$ which decomposes an object in A and determines its B component (assuming there is exactly one such component), and similarly for $s-c$, $s-d$, etc. Thus with the given definition for A:

$$a \in A \Rightarrow s-c.a = hd.tl.a .$$

3. HIERARCHIES

In characterizing an IMS the structure of the information in its data repository is specified first and the operations on that structure next.

As fundamental information structure we use the *hierarchy*. This is a tree having data elements (integers, strings, or entities) as leaves. All of its nodes (both leaves and branch points) are associated with a set of attributes, each attribute being a pair of a name and a value. The following domains will be used:

M *Atoms* or data elements are objects which are indivisible from the point of view of this theory.

E *Entities* are atoms which are used for fixed purposes in the information structure. They will be written like identifiers in programming languages, e.g. *john*, *red*, *linenumber*. We have $E \subset M$. The distinguished object *nil* is a member of M but not of E, and is identified with the empty sequence and the empty set. Integers and strings are also examples of atoms which are not entities.

The composite domains: T (trees), H (hierarchies), A (attributes), and L (labels), are defined by the following equations:

$$H = \{h\} \times L \times (T \cup M)$$
$$T = H^*$$
$$L = E \rightsquigarrow M \mid nil$$
$$A = E \times M .$$

Here h is a symbol which is not otherwise used. It serves as a mark on each hierarchy. The definitions mean that:

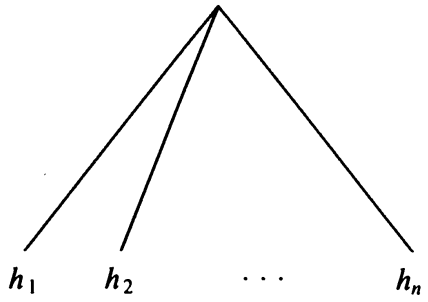
A *tree* is a sequence of hierarchies. In particular, *nil* is a member of T.

A *hierarchy* is formed from two elements, where the first one is a label and the second element is either an atom or a tree.

A *label* is a pseudo-mapping from entities to atoms, i.e. a certain set of attributes. (In other papers on the *use* of IMS we shall in fact need non-atomic attribute values in labels, but the above definition is sufficient for our present purpose).

An *attribute* is formed from two elements, where the first one is an entity and the second one is an atom.

We shall use a graphical notation for these structures, where atoms are written as text. A tree which is a sequence of hierarchies is written as follows (figure 1):



with the members of the sequence at the lower ends of the successive arcs (from left to right, of course).
A hierarchy is written as follows (figure 2):

where the box represents the label and provides space for writing (some of) the attributes.

Hierarchies may be written as formulas, using the notation that was introduced in the previous section, but we also introduce the following infix notation for improved legibility:

: is used as an infix symbol for forming hierarchies:
$J \in L \wedge x \in (T \cup M) \Rightarrow J : x = <h, J, x>$
and also for forming attributes:
$e \in E \wedge m \in M \Rightarrow e : m = <e, m>$.

; is used as an infix symbol for the function *pfx*, restricted to the domain $H \times T$, for constructing trees.

For example, $x ; y ; z ; nil = <x, y, z>$.
If h_1, h_2 and h_3 are hierarchies and J is a label,

$$J : (h_1 ; h_2 ; h_3 ; nil)$$

is another hierarchy.

We use the following *precedence rules* for the infix operators:

$x ; y ; z = x ; (y ; z)$
$J : x ; y = (J : x) ; y$
$x ; J : y = x ; (J : y)$
$a.b.x = a.(b.x)$.

4. SURROUNDINGS

The concept of a *cursor* is fundamental in most practical information management systems. It is a point in the data repository relative to which the operations are performed. We introduce the cursor as a distinguished object, which shall be written # . It is not a member of any of the domains that were introduced in the previous section. We shall use two domains defined informally as follows:

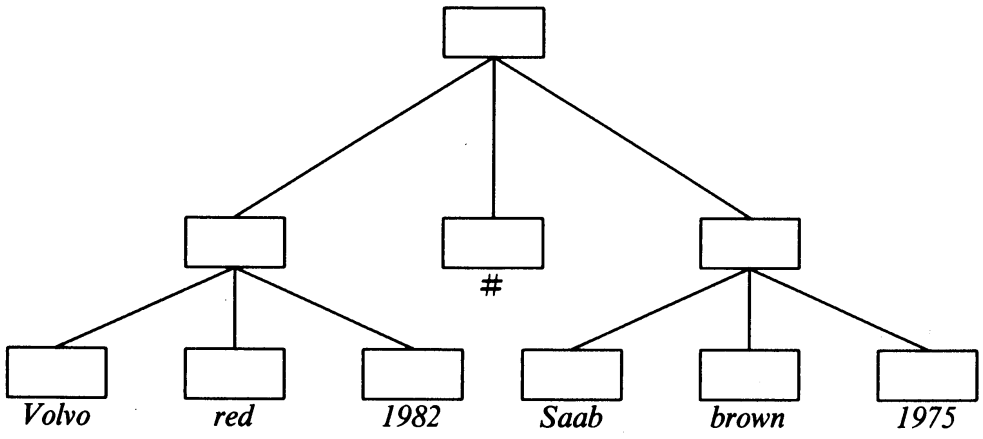
A *surrounding* is similar to a hierarchy, except that # occurs exactly once in a position which would otherwise have been taken by an atom or tree. The cursor may not occur in an attribute.

A *perspective* is also similar to a hierarchy, except that # occurs in exactly one position which would otherwise have been taken by a hierarchy.

In other words, the difference between surroundings and perspectives is that in a surrounding the cursor # 'has' a label (there is a substructure consisting of a label and the cursor). The cursor with 'its' label (resp. the cursor itself as the case may be) may be located between two hierarchies in a tree (= sequence of hierarchies). Thus it is located *between* structures rather than *at* a structure.

The surrounding is a basic concept in the IMS: the operations which the user invokes interactively while he is working with the system, such as inserting or deleting at the position of the cursor, or moving the cursor, are functions from surroundings to surroundings.

The following is an example of a surrounding (figure 3):



For a strict definition, we introduce variants of the recursively defined domains H and T as follows:

$$H' = \{h\} \times L \times (T' \cup \{\#\})$$
$$T' = H^* \ conc \ H'! \ conc \ H^*$$

and

$$H'' = (\{h\} \times L \times T'') \cup \{\#\}$$
$$T'' = H^* \ conc \ H''! \ conc \ H^* \ .$$

Thus every member of T' is a sequence of surroundings, exactly one of which contains the cursor symbol (not more than one of it) and similarly for T''. (Remember that $A! = \{<a> \mid a \in A\}$.) Then H' is the domain of surroundings and H'' is the domain of perspectives. We shall write

$U = H'$
$P = H''$.

For the specification of operations on surroundings it is convenient to introduce functions allowing us to describe a surrounding relative to the cursor position, so that the structures that are adjacent to the cursor appear close to the top of the expression rather than deep down in a substructure. We introduce the following functions:

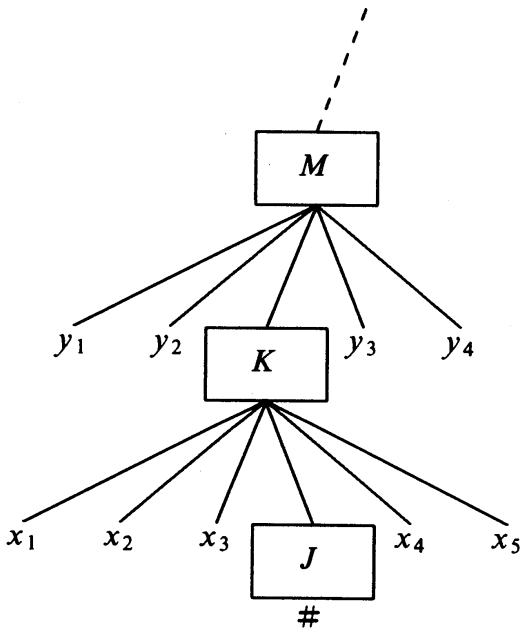
: $L \times P \rightarrow U$
$J:p = subst[\#, J:\#, p]$

per $T \times U \times T \rightarrow P$
$per[l,u,r] = subst[\#, rev.l \ conc \ \# ; r, u]$.

It is easily seen that the ranges of these functions are U and P, respectively. A surrounding as in figure 4 can now be written

$$J:per[l, K:p, r]$$

where r is the sequence of 'sister' hierarchies to the right of the cursor, in their ordinary order; J is the label just 'above' the cursor symbol in the diagrams; l is the sequence of 'sister' hierarchies to the left of the cursor, in reverse order; and K is the label immediately above J in the diagram. It is the label above the members of l and r in the hierarchy from which the surrounding was formed. Finally, p may be # or a new expression formed using *per*. In this way, a surrounding can be written so that the information close to the cursor appears on the top level of the expression. This is important when specifying IMS operations that have effects close to the cursor.



$J:per[l, K:p, r]$
where
$r = <x_4, x_5> = x_4; x_5; nil$
$l = <x_3, x_2, x_1> = x_3; x_2; x_1; nil$
$p = per[y_2; y_1; nil, M:p', y_3; y_4; nil]$

Figure 4

In the applications, attributes are used for a number of purposes: for specifying the field names for fields in a record; for specifying record types and the choice of keys; for specifying the position of various substructures on the screen or in a printout, etc. But there will also be many situations in which there are no attributes, i.e. in which the label is the empty set.

An expression $J:p$ is *fully inverted* if and only if either p is the constant # or if it has the form $per[l,u,r]$, where u is fully inverted. It is easily seen that each member of U can be written as a fully inverted expression in exactly one way.

Let us give one brief example of how this structure may be used. A conventional record may be represented by a hierarchy in which each daughter has the label

$$\{fld:n\}$$

(where *fld* is a constant and n is a variable). Such an expression will be abbreviated by capitalizing the symbol for n, for example

$$Year=\{fld:year\}\ .$$

The surrounding in figure 3 above, if provided with reasonable field names, can now be written:

$$nil:per[(Manuf:Volvo;Color:red;Year:1982;nil);nil,nil:\#,\\ (Manuf:Saab;Color:brown;Year:1975;nil);nil]\ .$$

If the cursor is moved to a position between the nodes for *Volvo* and *red*, the surrounding becomes:

$$nil:per[Manuf:Volvo;nil,\\ nil:per[nil,nil:\#,\\ (Manuf:Saab;Color:brown;Year:1975;nil);nil],\\ Color:red;Year:1982;nil]\ .$$

5. DEFINITIONS OF OPERATIONS AND THE APPROACH TO THEIR COMPILATION

The simple interactive operations in an IMS are those which add, delete, and modify structures immediately before or after the cursor and those which move the cursor to a new position, for example one step forward or backward. Thus a simple view of an IMS is that it is a system which at each moment has a *state* which is a member of U. It receives from the user successive operations which are mappings $U \rightarrow U$ and changes state accordingly. These operations can be conveniently specified using the notation of the previous sections. For example, the operation *nx* that moves the cursor one step to the right is characterized by the axiom:

$$nx.C:per[l,u,x;r]=C:per[x;l,u,r]$$

plus one other axiom which specifies what happens when the cursor is already at the right end of the sequence, and which for example may be chosen as:

$$v=C:per[l,u,nil]\Rightarrow nx.v=v\ .$$

When a command driven system such as an IMS is implemented, it is natural to organize the program around a *case* statement which contains one branch for each possible operation, each branch being the implementation of the axiom(s) specifying the

corresponding operation. The topic of the present paper is to show how the transition from specification to implementation for an operation can be done within a single logical system.

Before we go into the details of compilation, let us specify a number of additional operations which have been implemented in the present system in order to provide some intuition for what the specifications may look like. We only specify the main case in the definition, corresponding to the first line in the definition of *nx* above, and omit the specifications of exceptional cases.

The *bk* operation moves the cursor one step backward:

$$bk.C:per[x;l,u,r]=C:per[l,u,x;r]\ .$$

The *dw* operation travels downward along the substructure to the right of the present cursor position:

$$x\in T \Rightarrow dw.C:per[l,u,J:x;r]=C:per[nil,J:per[l,u,r],x]\ .$$

The *up* operation leaves a substructure and positions the cursor after it:

$$up.C:per[nil,J:per[l,u,r],x]=C:per[J:x;l,u,r]\ .$$

The *rs* operation resets or 'rewinds' the cursor to the beginning of the present subtree level:

$$rs.C:per[x;l,u,r]=rs.C:per[l,u,x;r]$$
$$v=C:per[nil,u,r]\Rightarrow rs.v=v\ .$$

The $in[x]$ operation inserts an element before the cursor:

$$in[x].C:per[l,u,r]=C:per[nil:x;l,u,r]\ .$$

The *del* operation deletes the element immediately after the cursor:

$$del.C:per[l,u,x;r]=C:per[l,u,r]\ .$$

We can now proceed to the issue of compilation. Our method is based on making a transformation between two subsets of the language of first-order predicate calculus (FOPC), namely the subset used for specifications and a subset which corresponds to a conventional programming language. The *specification* for an operation *op* is assumed to be written in the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

where each of the A_i is a *specification clause* having the form

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m \Rightarrow op.x=y$$

where x and y are surrounding-valued *expressions* and where the *literals* P_i are formed using the notation that was introduced in previous sections.

As the recursive specification of *rs* above shows, *op* may be used also for forming the expression y. Of course, other operations, defined in one or more other clauses of the specification, may occur as well. We do not impose such constraints as would guarantee in general that a specification is solvable; that has to be verified separately for each individual specification. The same applies to the programs that implement the specification.

The *procedure* for *op*, by contrast, is restricted to the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

where each A_i is a procedure clause of the form

$$P_1 \wedge \ldots \wedge P_n \Rightarrow op.u^0 = u^n$$

where u^0 and u^n are now restricted to *single variables* and where there are also a number of restrictions on the literals P_i. They may for example have the form

$$u^{k+1} = o.u^k$$

where u^{k+1} and u^k are single variables and o is an expression. The conversion from specification to program is a kind of pattern compilation: whereas the specification uses construction functions such as *per*, the program contains e.g. tests for membership in a domain and decomposition functions such as *rg*, defined by

$$rg.C : per[l,u,r] = r \ .$$

The language for procedure clauses resembles a simple, fairly conventional programming language (more precisely: a single assignment language), while it is at the same time represented entirely within FOPC. Various kinds of P_i correspond to various kinds of programming language statements: variable assignments, operations on the data base, conditionals, etc.

The representation of the procedure for an operation will be introduced in a stepwise fashion, by defining a sequence of sublanguages (each sublanguage being a first-order logic with certain syntactic restrictions).

6. DECOMPOSITION AND MODIFICATION FUNCTIONS

In the sublanguages we need the following predicates:

On T $\quad ispfx[t] \equiv (\exists x,y)\, t = x;y$
(In other words, t is not the empty sequence. The name for the predicate was selected because $x;y$ is also written $pfx[x,y]$.)

On P $\quad isper[p] \equiv (\exists l,u,r)\, p = per[l,u,r]$
(In other words, p is not *.)

On U $\quad nontop[C:p] \equiv isper[p]$.

We also need decomposition functions for each domain except M and its subsets. For the domains defined in section 3 this can be done by the ordinary conventions:

For H: if $h = J:x$, then
$j = s - l.h$
$x = s - tm.h$ (with a simple generalization of the name convention for selection functions).

For T: if $t = h_1; h_2; \ldots$, then
$h_1 = hd.t$
$h_2; \ldots = tl.t$. (The functions *hd* and *tl* are therefore in T restricted to those t that satisfy $ispfx.t$.)

For L, the . primitive serves to identify one component of the label at a time.

For A: if $a = e:m$, then
$e = s - e.a$
$m = s - m.a$.

When objects in the domains P and U are written using fully inverted expressions, the convention for forming decomposition functions of the form $s -$ do not apply as usual, since the functions : and *per* do not directly correspond to composition rules of the abstract syntax for these domains. However, since each surrounding has just one representation as a fully inverted expression, similar functions are well defined, e.g. the 'right list of sisters function',

$$rg.J:per[l,u,r]=r \ .$$

We shall use the mnemonic names *lbl* (label), *pc* (perspective content), *lf* (left), *ow* (owner), and *rg* (right) for decomposing a surrounding according to the definitions in the following table. The table also defines the modification functions *slf* (set left), *srg* (set right), *slbl* (set label), and *embed*. These functions are used for obtaining the state transitions required by an operation. For each function we specify its name, the types of its domain and range, the precondition under which it is defined (as an additional restriction on the domain) and, on the next line, the equation that specifies the relationship between argument and value.

name	**type**	**precondition**
lbl	$U \to L$ $lbl.J:p = J$	
pc	$U \to P$ $pc.J:p = p$	
rg	$U \to T$ $rg.J:per[l,u,r]=r$	*nontop*[*u*] required for *rg.u*
lf	$U \to T$ $lf.J:per[l,u,r]=l$	*nontop*[*u*] required for *lf.u*
ow	$U \to U$ $ow.J:per[l,u,r]=u$	*nontop*[*u*] required for *ow.u*
slf	$T \to (U \to U)$ $slf[t].C:per[l,u,r]=C:per[t,u,r]$	*nontop*[*u*] required for *slf*[*t*].*u*
srg	$T \to (U \to U)$ $srg[t].C:per[l,u,r]=C:per[l,u,t]$	*nontop*[*u*] required for *srg*[*t*].*u*
slbl	$L \to (U \to U)$ $slbl[J].C:p = J:p$	
embed	$L \times T \times T \to (U \to U)$ $embed[J,l,r].u = J:per[l,u,r]$	

The following properties of these functions are readily inferred from the definitions (all constrained by the preconditions of the functions):

$$nontop[srg[x].u]$$
$$nontop[slf[x].u]$$
$$nontop[u] \Rightarrow nontop[slbl[J].u]$$
$$nontop[embed[J,l,r].u]$$

$$lf.srg[x].u = lf.u$$
$$lf.slf[x].u = x$$
$$lf.slbl[J].u = lf.u$$

$$rg.srg[x].u = x$$
$$rg.slf[x].u = rg.u = rg.slbl[J].u$$

$$lbl.srg[x].u = lbl.u = lbl.slf[x].u$$
$$lbl.slbl[J].u = J$$

$$ow.srg[x].u = ow.u = ow.slf[x].u = ow.slbl[J].u$$

$$lf.embed[J,l,r].u = l$$
$$rg.embed[J,l,r].u = r$$
$$lbl.embed[J,l,r].u = J$$
$$ow.embed[J,l,r].u = u$$

These definitions should be included as proper axioms in a forthcoming first-order IMS theory, together with e.g. axioms which effectively are translations of the domain equations, such as

$$r \in T \Rightarrow ispfx[r] \vee r = nil \ .$$

Although the precise formulation of such a theory must wait for a later occasion, we can already observe here that, in the IMS theory, the relation between the specification of an operation and a procedure that implements it should be one of logical consequence, i.e. the implementation should be a consequence of the specification.

7. LPSL - LINEAR PROGRAM SUBLANGUAGE

An *LPSL program* is conjunction of *LPSL clauses*, each of which has the form

$$(\forall\, v^1, v^2, \ldots, v^m, u^0, u^1, \ldots, u^n)$$
$$P_1 \wedge P_2 \wedge \ldots \wedge P_p \Rightarrow op[v^1, \ldots, v^h].u^0 = u^n$$

The operation *op* is said to *have* this clause. The clause must satisfy:

I. Simple constraints:

$$m, n, p \geqslant 0$$
$$0 \leqslant h \leqslant m \ .$$

II. The literals P_i must have one of the following forms:

1. $u^k = o.u^{k-1}$

where o is an expression for a mapping $U \rightarrow U$, formed using one of the functions *ow*, *slf*, *srg*, *slbl*, *embed*, which have just been defined, or operations which *have* other

clauses in the same LPSL program.

2. $v^k = x$
where x is an expression whose value has a type other than U or P, and which is formed using composition and/or decomposition functions as defined above.

3. An arbitrary predicate expression z, formed using some of the predicates defined above (*ispfx*, *isper*, *nontop*) or an ordinary predicate such as equality.

The function *per* may not be used to form expressions, i.e. surroundings can only be modified, not created afresh. Also, the expressions x and z in cases 2 and 3 may not use any surrounding valued function except *ow*.

III. The constituent expressions (o, x, z) must satisfy certain constraints which we shall now define. For a given LPSL clause, with the variables and indices specified above, we define a sequence $c_0, c_1, \ldots, c_p$ of *current variable sets*, which are sets of variable symbols, (not sets of the objects that the variables denote), and which are defined as follows:

$c_0 = \{v^1, \ldots, v^h, u^0\}$;

if P_i has the form $u^k = o.u^{k-1}$ and
$c_{i-1} = c \cup \{u^{k-1}\}$, then
$c_i = c \cup \{u^k\}$;

if P_i has the form $v^k = x$, then
$c_i = c_{i-1} \cup \{v^k\}$;

if P_i is a predicate expression z, then
$c_i = c_{i-1}$.

Clearly every c_i contains exactly one u^k variable. The constraints on the sequence of P_i are now the following:

a) If P_i is of type (1) above, u^{k-1} must be a member of c_{i-1}. Intuitively speaking, u^k stands for the state of the data repository after the 'operation' P_i. The succession of such states as are obtained by successive update operations are presented in our logic by a sequence of u_k variables. The actual implementation in a conventional programming language can contain a single variable u and perform successive assignments to it and/or perform successive operations with side-effects on its value.

b) If P_i is of type (2) above, the variable to the left of the equality sign must not be a member of c_{i-1}. In other words, it must be a new addition to c_i. Intuitively, P_i is an assignment to a (programming language) variable to which no previous assignment has been made.

c) All variables which occur in o, x, or z (as the case for P_i may be) must be members of c_{i-1}. In programming language terms, this means that variables must have a value assigned to them before they are used.

If P_i is of type (3) i.e. a condition z, it should be viewed as the condition in an *if* statement (of a conventional programming language). The subsequent $P_{i+1} \ldots$ constitute the *then* branch of the *if* statement. The *else* branch may be specified by another LPSL clause.

d) A decomposition function as described above may be used in P_i only if its precondition occurs among or if it is a consequence in the IMS theory from

$P_1 \wedge P_2 \wedge \ldots \wedge P_{i-1}$. This is our counterpart of type checks in programming languages.

e) $u^n \in c_p$.

In order to make condition (d) effectively decidable in general, it would have to be strengthened to for example 'can be inferred in at most 1000*p deduction steps from...'. This practical matter will be bypassed here. This ends the list of constraints.

For example, the following is an LPSL clause:

$$
\begin{array}{l}
(\forall\, v^1, u^0, u^1, u^2) \\
\quad nontop[u^0] \vee \\
\quad v^1 = rg.u^0 \vee \\
\quad ispfx[v^1] \vee \\
\quad u^1 = slf[hd[v^1]; (lf.u^0)].u^0 \vee \\
\quad u^2 = srg[tl[v^1]].u^1 \Rightarrow nx.u^0 = u^2 .
\end{array}
$$

If an IMS theory is designed in the way suggested above, this LPSL clause must be a consequence in the theory of the following specification clause

$$nx.C : per[l, u, x; r] = C : per[x; l, u, r] .$$

To verify that constraint (d) on LPSL clauses is satisfied throughout, we notice that the precondition $nontop[u^0]$ legitimizes the use of $rg.u^0$, $lf.u^0$, and $slf.[\ldots].u^0$. The precondition $ispfx[v^1]$ legitimizes the use of $hd[v^1]$ and $tl[v^1]$. Finally, $u^1 = slf[\ldots].u^0$ implies $nontop[u^1]$ (above) which legitimizes $u^2 = srg[\ldots].u^1$.

The idea behind the LPSL clause is that, in addition to being a consequence of a specification clause, it should express a procedure for executing the operation in the cases covered by the specification clause. In other words, there should be a simple, essentially syntactic, transformation transforming an LPSL clause into a *reasonable* procedure in a conventional programming language doing the same thing. The intuition as to how the LPSL constructs are related to programming language constructs has already been given. The constraints that have been imposed on LPSL clauses were dictated by this goal. On the other hand, we are not yet ready to give a strict proof of this correspondence between LPSL and programming languages. The set of constraints that has been given here should therefore be seen as provisional. However, for the simple examples in this paper it should be fairly clear how a specification clause can be transformed into an LPSL clause and how an LPSL clause can be transformed into a procedure.

8. BPSL - BRANCHING PROGRAM SUBLANGUAGE

The specification of an operation may consist of several clauses, e.g.

$$
\begin{array}{l}
nx.C : per[l, u, x; r] = C : per[x; l, u, r] \wedge \\
nx.C : per[l, u, nil] = C : per[l, u, nil] .
\end{array}
$$

When these are transformed into LPSL, we often obtain LPSL clauses in which the first few A_i are the same. It is natural to introduce a sublanguage in which these can be shared and which thus contains the counterparts of *if* statements in conventional programming languages. This is what BPSL does. The present section presents BPSL, although still in a somewhat sketchy way.

A *BPSL program* is a set of BPSL procedures for different operations. A *BPSL procedure* for an operation *op* has the form

$$(\forall\, v^1, v^2, \ldots, v^m, u^0, u^1, \ldots, u^n)\ op[v^1, \ldots, v^h].u^0 = /\ R$$

where R is a *computation rule* in BPSL having the form

$$P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_p \rightarrow u$$

with $p \geqslant 0$, while the successive P_i satisfy the same conditions as in LPSL and u is either a variable u^m with $m \leqslant n$ or an expression

branch
if R_1
if R_2
...

Each of the R_i is called an *if* clause and again is a computation rule in BPSL. When several *if* clauses are nested inside each other, the possible ambiguity is resolved by indentation: *if* clauses in the same u have their initial *if* keyword directly underneath each other.

We must also impose on computation rules in BPSL a syntactic restriction corresponding to the restriction on LPSL clauses, but in order to make it understandable we must first make clear the meaning of BPSL procedures.

First, an example of a BPSL procedure:

$(\forall\, v^1, u^0, u^1, u^2)\ nx.u^0 = /$
  nontop $[u^0] \rightarrow$
  $v^1 = rg.u^0 \rightarrow$ *branch*
    if ispfx $[v^1] \rightarrow$
      $u^1 = slf\,[hd[v^1]; (lf.u^0)].u^0 \rightarrow$
      $u^2 = srg\,[tl[v^1]].u^1 \rightarrow u^2$
    if $v^1 = nil \rightarrow u^0$

This procedure is equivalent to the specification of *nx* that was given at the beginning of this section. The first *if* clause handles the case of the first line in the specification, while the second *if* clause handles the second line. Notice also that the second *if* clause uses u^0, not u^2, i.e. variable numbers are incremented in parallel in the branches.

The meaning of a BPSL procedure is defined by transformations to an equivalent set of LPSL clauses, as follows:

If v is a variable,

$$op.u\ = /\ v$$

has the same meaning as

$$op.u = v\ .$$

The expression

$$x = / P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_p \rightarrow u$$

has the same meaning as

$$P_1 \wedge P_2 \wedge \ldots \wedge P_p \Rightarrow x\ = /\ u\ .$$

The expression

$$x \; =/ \; branch$$
$$\quad if\, u^1$$
$$\quad if\, u^2$$
$$\quad \ldots$$

has the same meaning as

$$(x \; =/ \; u_1) \wedge (x \; =/ \; u_2) \wedge \ldots .$$

Using these rules, it is clearly possible to rewrite each BPSL procedure as a conjunction of LPSL clauses (apart from the constraints). We shall call this the *flat form* of the BPSL procedure. The above example of the BPSL procedure for *nx* can thus be rewritten as the conjunction of two LPSL clauses, one of which is identical to the clause for *nx* that was given in the previous section. We impose on BPSL procedures the natural constraint that each of the terms in the flat form must satisfy the constraints imposed on LPSL clauses.*

If a BPSL procedure is to be a usable program, it must take into account all cases that may occur in the IMS in which it is used. The completeness criterion is that in any expression

$$P_1 \rightarrow \ldots \rightarrow P_i \rightarrow branch$$
$$\quad if\, Q_1 \rightarrow v^1$$
$$\quad \ldots$$
$$\quad if\, Q_n \rightarrow v^n$$

the various criteria Q_i must satisfy (again in the IMS theory)

$$P_1 \wedge \ldots \wedge P_i \Rightarrow (Q_1 \vee \ldots \vee Q_n) .$$

The simplest way of satisfying this condition is if $n = 2$ and $Q_2 = \neg\, Q_1$, which means we effectively have an ordinary *if ... then ... else ...* expression. A more common case seems to be that the Q_i represent the various possible cases in the syntax for an element, e.g. whether a perspective is # or not, or (in the last example) whether a sequence is empty or not.

In our BPSL procedure for *nx* above, the property $nontop[u^0]$ is required by subsequent operations. It means the procedure is not complete in this sense. However, all IMS operations defined in section 5 assume the predicate *nontop* to be valid for their arguments and preserve that property. It would therefore be reasonable to treat it as an invariant of the IMS and to require only that BPSL procedures shall be well defined and complete relative to the invariant. (This relaxes requirement III.(d) in the definition of LPSL.)

---

* except, since different *if* clauses may require a different number of *u* variables, a clause in the flat form may end on

$$\cdots \Rightarrow op[v^1, \ldots, v^h].u^0 = u^k$$

where $k \leqslant n$ without necessary equality. Compare with the definition of LPSL clauses above.

9. SECOND EXAMPLE

The operation *up* may be specified as:

$$up.J:per[nil,K:per[ll,uu,rr],r]=J:per[(K:r);ll,uu,rr] \wedge$$
$$up.J:per[nil,K:nil,r]=J:per[nil,K:nil,r] \wedge$$
$$up.J:per[x;l,u,r]=up.J:per[l,u,x;r]$$

and a derived expression in BPSL is:

$$(\forall\, v^1,v^2,u^0,u^1,u^2,u^3)\ up.u^0 = /$$
$$\quad nontop[u^0] \rightarrow branch$$
$$\quad if\ ispfx[lf.u^0] \rightarrow$$
$$\quad\quad u^1=bk.u^0 \rightarrow$$
$$\quad\quad u^2=up.u^1 \rightarrow u^2$$
$$\quad if\ lf.u^0=nil \rightarrow$$
$$\quad\quad v^1=lbl.u^0 \rightarrow$$
$$\quad\quad v^2=rg.u^0 \rightarrow branch$$
$$\quad\quad if\ nontop[ow.u^0] \rightarrow$$
$$\quad\quad\quad u^1=ow.u^0 \rightarrow$$
$$\quad\quad\quad u^2=slf[(lbl.u^1):v^2;(lf.u^1)].u^1 \rightarrow$$
$$\quad\quad\quad u^3=slb[v^1].u^2 \rightarrow u^3$$
$$\quad\quad if\ pc[u^1]=nil \rightarrow u^0$$

where *bk* was defined in section 5 as the inverse operation of *nx*.

10. IPSL - THE IMPLICIT-SURROUNDING PROGRAM SUBLANGUAGE

LPSL and BPSL use explicit variables $u^0,u^1,\ldots$ for the successive surroundings that are the states of the machine. When an expression in these sublanguages is transformed into a conventional program, as is readily done, it is of course possible to use a single variable for the 'current u' in the implementation, but the notation is still unnecessarily cluttered by all the u variables. An implicit-surrounding program sublanguage IPSL, in which references to the u^i surroundings do not have to be written out, can be defined by a reversible transformation from BPSL to IPSL.

Each BPSL procedure

$$(\forall\, v^1,\ldots,v^m,u^0,\ldots,u^n)\ op[v^1,\ldots,v^h].u^0 = /\ R$$

is transformed into

$$Op[v^1,\ldots,v^h] = /\ (\textbf{local}\ v^{h+1},\ldots,v^m)\ R'$$

where R' is obtained from R by the *is* transformation.

A computation rule in BPSL,

$$P_1 \rightarrow \ldots \rightarrow P_p \rightarrow u$$

is transformed into

$$P_1' \rightarrow \ldots \rightarrow P_p'$$

if u is a single variable, and into

$$P_1' \rightarrow \ldots \rightarrow P_p' \rightarrow u'$$

if u is a branch expression. In the latter case, the *if* clauses in the branch expression are all transformed using the same *is* transformation.

A literal P_i in a computation rule is transformed into a corresponding literal P_i' by the following transformation, according to the various cases that were specified in section 7:

An expression

$$u^k = x$$

where x has the form $o.u^{k-1}$, is transformed into an expression

$$x'$$

where the transformation from x to x' will be specified immediately.

An expression

$$v^k = x$$

is transformed into an expression

$$v^k = x' .$$

A predicate expression z, finally, is transformed into an expression z'. In all cases, the transformation from x or z to x' or z' is defined recursively as follows:

Constants are unchanged. Variables v^j are also unchanged. Variables u^j for surroundings cannot be transformed.

Functions $U \rightarrow U$ with an explicit variable as argument are capitalized and the argument is omitted. Thus:

$$\begin{aligned}
&nontop[u] \rightarrow Nontop\\
&lbl.u \rightarrow Lbl\\
&pc.u \rightarrow Pc\\
&rg.u \rightarrow Rg\\
&lf.u \rightarrow Lf\\
&ow.u \rightarrow Ow\\
&slf[t].u \rightarrow Slf[t']\\
&srg[t].u \rightarrow Srg[t']\\
&slbl[J].u \rightarrow Slbl[J']\\
&embed[J,l,r].u \rightarrow Embed[J',l',r'] .
\end{aligned}$$

For those cases in which the argument to a function from U to U is not a variable symbol, we introduce the operator $\otimes$ defined by

$$f.g.x = (f \otimes g).x$$

so that e.g. $nontop[ow.u^0]$ can be rewritten as $Nontop \otimes Ow$.

Other functions and predicates (e.g. *ispfx*, *hd*, ;) retain their argument structure, but each of the arguments undergoes the same transformation.

Finally, we add some syntactic sugar: if the final step in a branch has the form

$$if\ Q \rightarrow u^m$$

where u^m is a single variable, then it may be rewritten more suggestively as

if $Q' \rightarrow$ *Ident*

rather than

if Q' .

For example, the definition of *nx* that was given in BPSL in section 8, will be rewritten as:

$Nx = /$ (**local** v^1)
  Nontop $\rightarrow$
  $v^1 = Rg \rightarrow$ *branch*
    if ispfx$[v^1] \rightarrow$
      Slf$[hd[v^1]; Lf] \rightarrow$
      Srg$[tl[v^1]]$
    if $v^1 = nil \rightarrow$ *Ident* .

In this way we have moved our notation closer to the notation found in conventional programming languages, but this is only a matter of syntax. The semantics of first-order logic is retained.

11. TRANSFORMATIONS TO STACK MACHINE SUBLANGUAGES

The sublanguages introduced in the previous sections correspond to conventional programming languages in the sense that their literals correspond to different kinds of 'statements' for variable assignment, operations on the data base, conditionals, etc. An application of this correspondence is a translator from (e.g.) BPSL to a conventional programming language. Such a translator only performs a syntactic transformation into a subset of the target language. In this way we obtain a transformation from a specification language to a programming language. The latter can be compiled for execution on a conventional machine. These transformations are illustrated in figure 5.

However, we could achieve greater conceptual economy by considering e.g. BPSL to be *the* programming language for our use. We must then apply conventional compilation techniques to BPSL. The present section will show how this can be done by an extension of the same strategy as was used in previous sections, i.e. by making transformations into yet another subset of first-order logic, which intuitively corresponds to the machine language for a stack machine.

We define the following domains:

$B = \{$**true, false**$\}$

$Q = B \cup M \cup H \cup T$

Q is the domain of practically everything. It is needed for defining counterparts of registers and stacks.

$S = U \times Q^* \times Q$

S is the domain of *states* for the stack machine. In a state $<u, d, q>$, the surrounding u is the present contents of the data repository, d is the present stack, and q is the present contents of a distinguished register. Whenever we want to refer to the contents of the top of the stack, we make an operation that pops the stack into the register and then refer to the present contents of the register. We shall discuss the need for such a

register below.

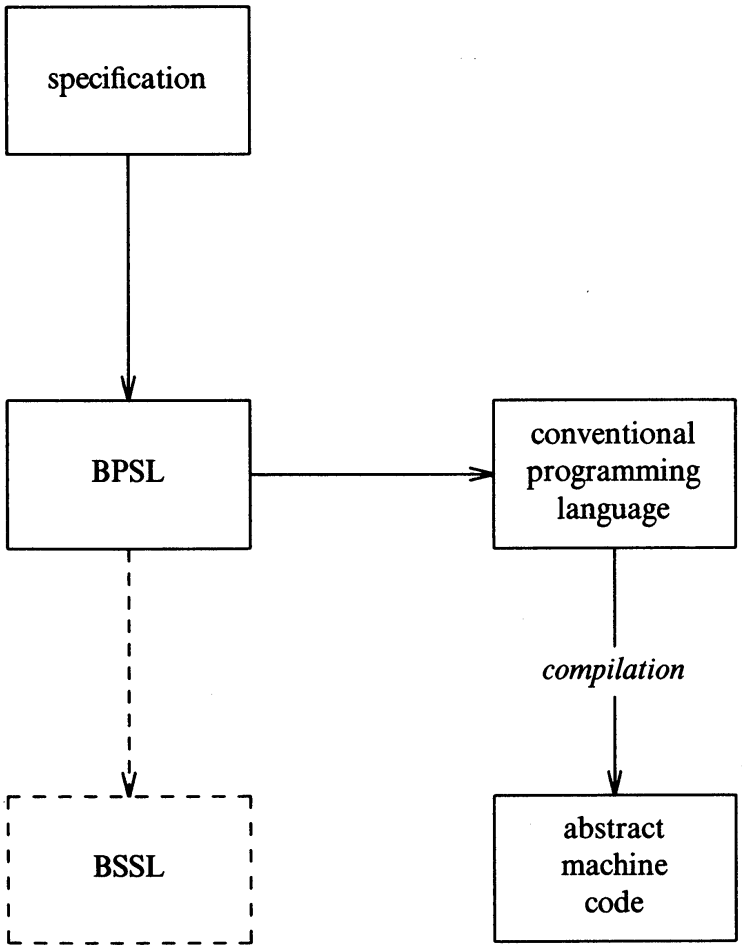


Figure 5

We shall now define a sublanguage, LSSL, which is analogous to LPSL except that it uses states whenever LPSL uses surroundings. As a consequence, it uses a set of operations on states which is different from (although related to) the operations on surroundings that are used in LPSL. Later on, the additional steps that are taken from LPSL to BPSL and IPSL will have direct counterparts for LSSL.

We define the following new predicates and functions on states:

name	**type**
push	$Q \rightarrow (S \rightarrow S)$ **push**$[x].<u,d,q>=<u,x;d,q>$
pop	$S \rightarrow S$ **pop**$.<u,x;d,q>=<u,d,x>$
reg	$S \rightarrow Q$ **reg**$.<u,d,q>=q$
test	predicate on S, defined as: **test**$.<u,d,q> \equiv q=$**true**

Furthermore, for the predicates and functions that were defined in previous sections with U as domain, we define bold-face counterparts on S. For the predicates as well as for those functions whose range is not U, the counterpart pushes an element on the stack:

nontop $nontop.u \Rightarrow \mathbf{nontop}.<u,d,q>=<u,\mathbf{true};d,q>$
$\neg\, nontop.u \Rightarrow \mathbf{nontop}.<u,d,q>=<u,\mathbf{false};d,q>$

lbl $\mathbf{lbl}.<u,d,q>=<u,(lbl.u);d,q>$

pc $\mathbf{pc}.<u,d,q>=<u,(pc.u);d,q>$

rg $\mathbf{rg}.<u,d,q>=<u,(rg.u);d,q>$

lf $\mathbf{lf}.<u,d,q>=<u,(lf.u);d,q>$.

For those functions whose range is U, the counterpart modifies the surrounding in the state:

ow $\mathbf{ow}.<u,d,q>=<ow.u,d,q>$

slf $\mathbf{slf}.<u,x;d,q>=<slf[x].u,d,q>$

srg $\mathbf{srg}.<u,x;d,q>=<srg[x].u,d,q>$

slbl $\mathbf{slbl}.<u,x;d,q>=<slbl[x].u,d,q>$

embed $\mathbf{embed}.<u,r;J;l;d,q>=<embed[l,J,r].u,d,q>$.

Predicates and functions with other domains than U are given boldface counterparts that take their arguments from the stack of the state and also return their value (represented using the domain B in the case of predicates) on the stack. For example, we define:

pfx $\mathbf{pfx}.<u,y;x;d,q>=<u,(x;y);d,q>$

hd $\mathbf{hd}.<u,x;d,q>=<u,(hd.x);d,q>$

ispfx $ispfx.x \Rightarrow \mathbf{ispfx}.<u,x;d,q>=<u,\mathbf{true};d,q>$
and a similar definition for the case of $\neg\, ispfx.x$.

equal $x=y \Rightarrow \mathbf{equal}.<x,y;x;d,q>=<u,\mathbf{true};d,q>$
and so forth.

Starting with an LPSL expression representing operations on surroundings, we can rewrite it as an LSSL expression representing operations on stacks by changing every u^k variable to a corresponding s^k variable, and by converting composite expressions into sequences of stack operations. For example, the expression

$$u^2=srg[tl[v^1]].u^1$$

(in the BPSL procedure for nx) is transformed into

$$s^2 = \mathbf{push}[v^1].s^1 \rightarrow$$
$$s^3 = \mathbf{tl}.s^2 \rightarrow$$
$$s^4 = \mathbf{srg}.s^3 .$$

The first operation pushes the value of v^1 on the stack; the second one performs a *tl* operation on the top element of the stack, and the third one pops the top element off the stack and inserts it as the new left sister list of the present surrounding.

More generally, the various types of literals in LPSL are transformed as follows:

Type (1) literals in LPSL are reduced to a sequence of literals in LSSL which first pushes elements on the stack with operations such as **push** with a single variable parameter, **rg**, **lf**, etc., later modifies them with operations that accept/put arguments/results from/on the stack, and finally deletes them with an operation such as **srg**.

Type (2) literals in LPSL are similarly transformed into a sequence of literals in LSSL, the last two clauses of which are:

$$s^{k+1} = \mathbf{pop}.s^k \rightarrow$$
$$v = \mathbf{reg}.s^{k+1} .$$

Finally, type (3) literals are transformed into sequences of LSSL literals ending on:

$$s^{k+1} = \mathbf{pop}.s^k \rightarrow$$
$$\mathbf{test}.s^{k+1} .$$

With this notation, we can e.g. rewrite the LPSL program for *nx* as

$$(\forall\, v^1, s^0, \ldots, s^{15})$$
$$s^1 = \mathbf{nontop}.s^0 \wedge$$
$$s^2 = \mathbf{pop}.s^1 \wedge$$
$$\mathbf{test}.s^2 \wedge$$
$$s^3 = \mathbf{rg}.s^2 \wedge$$
$$s^4 = \mathbf{pop}.^3 \wedge$$
$$v^1 = \mathbf{reg}.s^4 \wedge$$
$$s^5 = \mathbf{push}[v^1].s^4 \wedge$$
$$s^6 = \mathbf{ispfx}.s^5 \wedge$$
$$s^7 = \mathbf{pop}.s^6 \wedge$$
$$\mathbf{test}.s^7 \wedge$$
$$s^8 = \mathbf{push}[v^1].s^7 \wedge$$
$$s^9 = \mathbf{hd}.s^8 \wedge$$
$$s^{10} = \mathbf{lf}.s^9 \wedge$$
$$s^{11} = \mathbf{pfx}.s^{10} \wedge$$
$$s^{12} = \mathbf{slf}.s^{11} \wedge$$
$$s^{13} = \mathbf{push}[v^1].s^{12} \wedge$$
$$s^{14} = \mathbf{tl}.s^{13} \wedge$$
$$s^{15} = \mathbf{srg}.s^{14} \Rightarrow \mathbf{nx}.s^0 = s^{15}$$

We can now see why even for a stack machine the register component of the state is needed: for type (2) literals, we must pop a value from the stack and bind that value to a v variable, but at the same time we must bind the new state containing the popped stack to an s variable. This is taken care of by the register as shown in the example. Since we do not perform any operations on the contents of the register nor allow the register contents to be pushed back on the stack, we still have essentially a stack

machine.

Since LSSL is similar to LPSL, we can now introduce branch constructs into LSSL. This results in a sublanguage BSSL, in the same way as for BPSL. We must notice, however, that in BPSL it was the first literal after the *if* that constituted the test, whereas in BSSL the test (using the operation **test** defined above) can occur only after a number of operations involving the stack. It should be thought of as an exit from the *if* clause or, more precisely, an exit that resets the state of the machine to what it was when the *if* clause was entered.

The reserved word *if* in the BPSL syntax is therefore inappropriate and we shall use >> for the same purpose of marking the start of an *if* clause in BSSL.

Finally, we can eliminate the state variables from the notation by introducing ISSL by the same conventions as for IPSL, i.e. we capitalize functions and predicate names to indicate that a state parameter is implicit. At the same time, we allow comments to be written at the end of each line. We then have something which feels like the machine language of a stack machine, except for the way variables are handled:

Nx = / (*local* v^1)	
Nontop →	check whether *surr* is $C:per[l,u,y]$ rather than $C:\#$
	push truth value on stack
Pop →	pop truth value to register
Test →	if not satisfied, exit
Rg →	push y on stack
Pop →	move y to register
v^1=**Reg** → *branch*	bind v to y
>> **Push**[v^1] →	push y on stack
Ispfx →	check that y is not nil
Pop →	pop result of test to register
Test →	exit if test fails
Push[v^1] →	push y again, assume it is $x;r$
Hd →	change y into x on top of stack
Lf →	push l above x on top of stack
Pfx →	now $x;l$ is on top of stack
Slf →	assign $x;l$ as new list of left sisters in surrounding
Push[v^1] →	push $y=x;y$ on top again
Tl →	now r is on top of stack
Srg	assign r as new list of right sisters in surrounding;
	current state is result
>> **Push**[v^1] →	push y on stack again
Push[*nil*] →	push *nil* above it
Equal →	compare
Pop	
Test	if not equal then exit,
	otherwise current state is result.

The machine is a bit clumsy because of all the transfers to and from the register, but this can easily be remedied by introducing and using a few more operations. In principle we already have a machine here to which most of the intuitions of regular stack machines apply.

12. PROPOSED CONTINUED WORK

The present paper has been semiformal. The next step would be to verify that, for strict definitions of all languages involved, the transformations between the languages are possible in all cases. We have tried to convince the reader that such proofs will be possible, but there are some minor details which have been bypassed in the present paper. For example, we do not allow surroundings to be pushed on the stack. This is in principle a reasonable constraint, but as a compensation we must find some way of allowing the user to access the components of the owner of the current surrounding, for example in the expression

$$nontop\,[ow.u^0]$$

in the example in section 9. This requires some simple additions to the repertoire of operations.

The ISSL language, which was the last transformation step in the present paper, is not in all respects a reasonable machine language. A few more transformations and modifications should be made:

- Introduce a real register machine and/or smooth the stack/register transfers in the present treatment.
- Treat program variables in a variable stack in a separate component of the machine state rather than modeling them by predicate-logic variables.
- If, with the current definition of BSSL, the **test** statement fails during execution of an *if* clause an exit from the *if* clause should be performed and the next *if* clause should be entered with the machine in the state it was in when the first *if* clause was entered. In an implementation this would require that a copy of the state be made, but it is intuitively clear that the surrounding and stack at the time of exit are the same as at the time the first *if* clause was entered. In this and other ways, machine behaviour more like that of a real machine should be obtained, while retaining the close correspondence through all levels from specification to executable machine language.

REFERENCES

[BLI82] Blikle, A., "Desophisticating denotational semantics," to appear in: *Proceedings of the IFIP World Computer Congress*, 1983.

[BOE80] Boehm, B.E., "Developing small-scale application software products," **in** Lavington, S.H., (Ed.), *Information Processing 83*, North-Holland, 1980.

[SAN82] Sandewall, E., "An approach to information management systems," Report LiTH-MAT-R-82-19, Software Systems Research Center, Linköping University, July 1982.