LiTH-MAT-R-83-02

February 1983

DATALOGI LINKÖPING

UNIVERSITETET I LINKÖPING
TEKNISKA HÖGSKOLAN

RESEARCH REPORT

FORMAL SPECIFICATION AND IMPLEMENTATION OF OPERATIONS IN INFORMATION MANAGEMENT SYSTEMS

by

Erik Sandewall

Software Systems Research Center
Linköping institute of Technology
S-581 83 Linköping
Sweden

LiTH-MAT-R-83-02
Matematiska Institutionen
Tekniska högskolan i Linköping
ISSN 0348-2960

SOFTWARE SYSTEMS RESEARCH CENTER
Linköping Institute of Technology

PhD theses:

(Linköping Studies in Science and Technology Dissertations.)

No 14	Anders Haraldsson: A Program Manipulation System Based on Partial Evaluation, 1977.
No 18	Mats Cedvall: Semantisk Analys av processbeskrivningar i naturligt språk, 1977
No 22	Jaak Urmi: A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978
No 33	Tore Risch: Compilation of Multiple File Queries in a Meta-database System, 1978.
No 51	Erland Jungert: Synthesizing Database Structures from a User Oriented Data Model, 1980.
No 54	Sture Hägglund: Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980. (Contains LiTH-MAT-R-79-39, LiTH-MAT-R-80-01, LiTH-MAT-R-80-37, and LiTH-MAT-R-80-38)
No 55	Pär Emanuelson: Performance Enhancement in a Well-structured Pattern Matcher through Partial Evaluation, 1980.
No 69	H. Jan Komorowski: A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981.
No 71	René Reboh: Knowledge Engineering Techniques and Tools for Expert Systems, 1981.
No 77	Östen Oskarsson: Mechanisms of Modifiability in large Software Systems.

Research Reports published 1976-1983.

LiTH-MAT-R-76-8	Erik Sandewall: Some observations on conceptual programming. Also in Elcock, E.W., Michie, D. (eds), *Machine Intelligence 8*, Edinburgh University Press, 1977.
LiTH-MAT-R-76-9	Erik Sandewall: Programming in an interactive environment: The LISP experience. Also in *ACM Comp. Surveys, vol. 10*, no 1, pp 35 -71, march 1978.
LiTH-MAT-R-76-11	Anders Beckman: Programvarukvalitet, *En nordisk förstudie.*
LiTH-MAT-R-76-13	Sture Hägglund, Östen Oskarsson: En teknik för utformning av användardialoger i interaktiva datasystem.
LiTH-MAT-R-76-17	Jaak Urmi: String to string correction.
LiTH-MAT-R-76-18	Jaak Urmi: A shallow binding scheme for fast environment changing in a "spaghetti stack" LISP system.
LiTH-MAT-R-77-2	Jim Goodwin: A guided tour of the interlisp system
LiTH-MAT-R-77-26	Anders Haraldsson: A partial evaluator, and its use for compiling iterative statements in Lisp. Also in *Proc. of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, 1978.
LiTH-MAT-R-78-1	Anders Beckman, Sture Hägglund, Gunilla Lönnemark: A program package supporting run time variation of text output from interactive programs.
LiTH-MAT-R-78-19	Erik Tengvald, Sture Hägglund: En metadatabas för Cobolsystem.
LiTH-MAT-R-78-20	Jerker Wilander: Interaktiv programutveckling i Pascal - Programmeringssystemet Pathcal.
LiTH-MAT-R-78-21	Erik Sandewall: Programmeringsteknik för flexibilitet.
LiTH-MAT-R-78-22	Erik Sandewall: LOIS - An Overview of Facilities and Design. Also in *DATA, vol 9*, nr 1-2, February 1979. Revised version as "Provisions for Flexibility in the Linköping Office Information System", *Proc. of the National Comp. Conf.*, Los Angeles, 1980.
LiTH-MAT-R-78-23	Claes Strömberg, Henrik Sörensen: Beskrivning av fontsystemet och erfarenheter vid systemutvecklingen.
LiTH-MAT-R-78-24	Erland Jungert: Generering av databasstrukturer från en formulärbaserad datamodell.
LiTH-MAT-R-78-25	Pär Emanuelson: A case study of Qlisp: Representing knowledge taken from medical diaries.
LiTH-MAT-R-79-3	Arne Börtemark, Hans Lunell: Implementering av Pascal på minimaldatorn: en tillbakablick.
LiTH-MAT-R-79-5	Jim Goodwin: Taxonomic Programming with Klone.
LiTH-MAT-R-79-7	Pär Emanuelson: A comparative study of some pattern matchers.
LiTH-MAT-R-79-10	Arne Börtemark: Felbekämpningsmedel, en första översikt.
LiTH-MAT-R-79-18	Hans Lunell: Automatic generation of code for conceptual machines: A problem discussion.
LiTH-MAT-R-79-19	Jan Komorowski: QLOG interactive environment - the experience from embedding a generalized Prolog in INTERLISP.
LiTH-MAT-R-79-21	Erik Sandewall: Biological Software. Also in *Proc. of the 6th Int. Joint Conf. of Artificial Intelligence*, Tokyo, 1979.
LiTH-MAT-R-79-22	Erik Sandewall: Self-description and reproduction in distributed programming systems.
LiTH-MAT-R-79-23	Erik Sandewall: A description language and pilot-system executive for information-transport systems. Also in *Proc. of the 5th Int. Conf on Very Large Data Bases*, Rio de Janeiro, 1979.
LiTH-MAT-R-79-24	Erik Sandewall, Erland Jungert, Gunilla Lönnemark, Katarina Sunnerud, Ove Wigertz: A tool for design and development of medical data processing systems. Also in *Proc. of the 2nd Congress on Medical Informatics, Europe*, West Berlin, 1979.
LiTH-MAT-R-79-28	Erik Sandewall: Why superroutines are different from subroutines.
LiTH-MAT-R-79-37	Jerker Wilander: An interactive programming system for Pascal. Also in *BIT, vol 20*, 2, 1980.
LiTH-MAT-R-79-39	Sture Hägglund: An Application of Lisp as an Implementation Language for the Domain Expert's Programming Environment.
LiTH-MAT-R-79-42	Anders Ström: Experiment med partialevaluering.

(Continued on back cover.)

Lith - MAT-R-- 83-02.

FORMAL SPECIFICATION AND IMPLEMENTATION OF OPERATIONS IN INFORMATION MANAGEMENT SYSTEMS

Erik Sandewall

Software Systems Research Center
Linkoping University
Linkoping, 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, the 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.

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

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 householding': entering data into the computer, changing it; displaying a 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, and as Boehm has shown (ref. BOE80) the implementation of an interactive application program consists to a large extent in 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 for what this kind of software really does. In fact, the lack of formal understanding is probably one reason for the proliferation: various IMS perform 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 accomodate the application specific details as well as the generalitities that are present in all IMS.

In a previous paper (ref. SAN82A), 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 of most IMS, and a framework for characterizing application-specific services). In particular, the following things are characteristic for IMS, and are accounted for in that reference:

- 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, particularly the *common command loop*. An additional problem is the *prompting* situations 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 whereby the specification of operations (i.e. of their effect on the data repository) can be transformed into a procedure which executes the operation. 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 data repository of the IMS. Following the advise of Blikle (ref. BLI82), we shall express our domain equations in set theory notation, and 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
$\supset$	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
$\langle x,y,\ldots\rangle$	The sequence whose members are x, y, etc. $\langle x\rangle$ 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	$hd.\langle x_1, x_2, \ldots x_n\rangle = x_1$
tl	$tl.\langle x_1, x_2, \ldots x_n\rangle = \langle x_2, \ldots x_n\rangle$
pfx	$pfx(x, \langle x_1, \ldots x_n\rangle) = \langle x, x_1, \ldots x_n\rangle$
conc	Concatenation of sequences. We define $\langle x_1, \ldots x_k\rangle$ *conc* $\langle x_{k+1}, \ldots x_n\rangle = \langle x_1, \ldots x_n\rangle$ We immediately extend this operation to sets of sequences in the obvious way: A *conc* $B = \{a$ *conc* $b / a \in A \wedge b \in B\}$
rev	reversal of sequences
subst	substitution in a sequence structure, defined so that *subst[old,new,x]* replaces every occurence of *old* into an occurrence of *new* in the structure x
x	Cartesian product. We define $A \times B \times \ldots \times D$ as the set of all $\langle a, b, \ldots d\rangle$ where $a \in A$ etc.
!	Set of onetuples: $A! = \{\langle a\rangle / a \in A\}$
*	$A^* = \{\langle a_1, \ldots a_n\rangle \mid a_i \in A \wedge n > 0\}$
+	$A^+ = \{\langle a_1, \ldots a_n\rangle \mid a_i \in A \wedge n > 1\}$
$\xrightarrow[pm]{}$	pseudo-mapping: $A \xrightarrow[pm]{} B \mid e$ is the set of all total functions from A to B such that $f[a] \neq e$ for just a finite number of arguments.

If f is a pseudomapping for which e is *nil*, then we will confuse it with the set of all pairs $\langle x, f[x]\rangle$ where $f[x] \neq nil$. In particular, the pseudomapping which maps everything to *nil* is confused with the empty set.

Named domains and selector functions.

We use the symbols that have been introduced for writing domain equations, which then provide 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 -> B which decomposes an object in *A* and determines its *B* component (assuming there is exactly one such), and similarly for *s-c*, *s-d* etc. Thus with the given definition for *A*,
$a \in A \supset s\text{-}c.a = hd.tl.a$

3. Hierarchies.

The characterization of the IMS begins with the structure of information in its data repository, and proceeds then to characterize the operations on that structure.

As fundamental information structure we use the *hierarchy*, which is a tree where the leaves are data elements (integers, strings, or entities), and where all 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:

$$H = \{h\} \times L \times (T \cup M)$$
$$T = H^*$$
$$L = E \xrightarrow[pm]{} M \mid nil$$
$$A = E \times M$$

Here h is a symbol which is not otherwise used, and which 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.

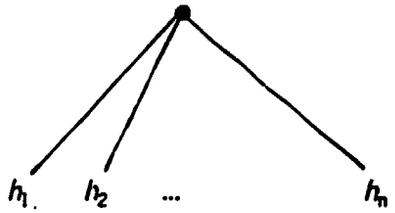
A *hierarchy* is formed from two elements, where the first one is a label, and the second element is either an atom or a tree.

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 will in fact need non-atomic attribute values in labels, but the present 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); and a hierarchy is written as follows (figure 2):

where the box represents the label, and provides a 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: $J \in \mathsf{L} \wedge x \in (\mathsf{T} \cup \mathsf{M}) \supset$ $J{:}x = \langle h, J, x \rangle$ and also for forming attributes: $e \in \mathsf{E} \wedge m \in \mathsf{M} \supset$ $e{:}m = \langle e,m \rangle$
;	is used as an infix symbol for the function *pfx*, restricted to the domain $\mathsf{H} \times \mathsf{T}$, for constructing trees.

For example, $x; y; z; nil = \langle x, y, z \rangle$. If h_1, h_2, and h_3 are hierarchies, and J is a label, then

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

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 •, and which 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:

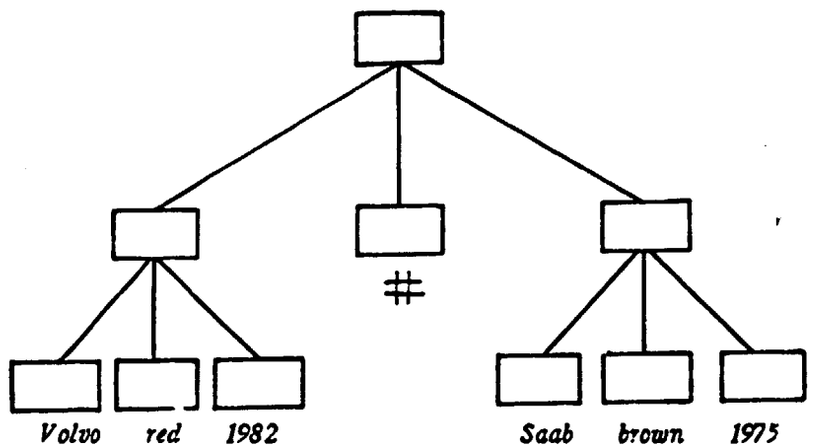
A ***surrounding*** is similar to a hierarchy, except that • occurs exactly once in a position which would otherwise have been taken by an atom or tree. The cursor may not occur in an attribute.

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 stands ***between*** structures, rather than ***at*** a structure.

The surrounding is a basic concept in the IMS: the operations that the user invokes interactively as 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 9):



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

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

and

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

Thus every member of T' is a sequence of surroundings, exactly one of which contains the cursor symbol, and it only contains it once, and similarly for T''. (Remember that $A! = \{\langle a \rangle / a \in A\}$). 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 which allow 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 sub-structure. We introduce the following functions:

: $L \times P \rightarrow U$

defined by:

$J:p = subst[\#, J:\#, p]$

per T x U x T -> P
defined by:
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 then 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 that is next above *J* in the diagrams, and which 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 that is located close to the cursor appears on the top level of the expression. This is important for us when we are going to specify IMS operations that have effects close to the cursor.

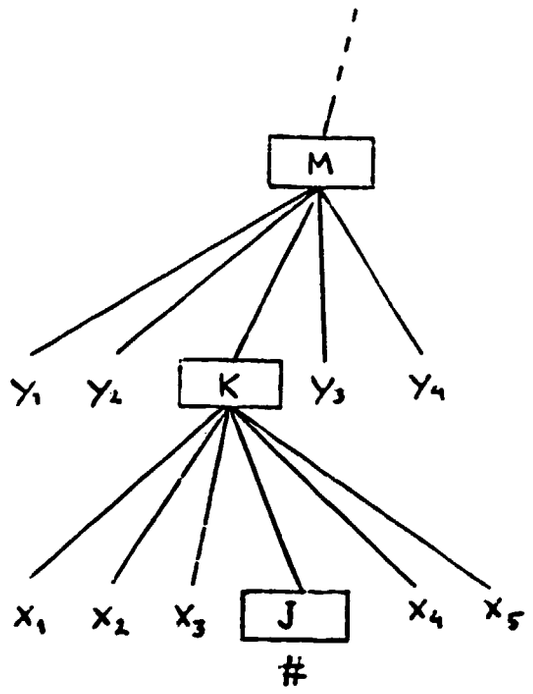
In the applications, attributes are used for a number of purposes: for specifying the field names for fields in a record; for specifying record types and the choice of keys; for specifying the position of various sub-structures on the screen or in printout; etc. But there will also be many situations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

An expression *J:p* is *fully inverted* iff either *p* is the constant #, or 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 where each daughter has the label
{fld:n}
(where *fld* is a constant, *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 is provided with reasonable field names, can then 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 between the nodes for *volvo* and *red*, the surrounding is
nil:per[Manuf:volvo;nil,
nil:per[nil, nil:#,
(Manuf:saab; Color:brown; Year:1975; nil); nil],
Color:red; Year:1982; nil]



J: per[ℓ, K:p, r]

where $r = \langle x_4, x_5 \rangle = x_4; x_5; nil$

$\ell = \langle x_3, x_2, x_1 \rangle = x_3; x_2; x_1; nil$

$p = per[y_2; y_1; nil,\ M:p',\ y_3; y_4; nil]$

Figure 4.

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 in each moment has a *state* which is a member of U, and which receives from the user successive *operations* which are mappings U -> 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 following axiom:

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

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

$v = C{:}per[l,u,nil] \supset nx.v = v$

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, and where that branch is then the implementation of the axiom(s) whereby the operation is specified. The topic of the present paper is to show how the transition from specification to implementation for an operation can be done within one 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 give 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 specifications of exceptional cases.

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

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

The *dw* operation which goes down into the sub-structure to the right of the present cursor position:

$x \in T \supset$

$dw.C{:}per[l, u, J{:}x;r] = C{:}per[nil, J{:}per[l,u,r], x]$

The *up* operation for leaving a sub-structure and positioning the cursor after it:

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

The *rs* operation for resetting or "rewinding" the cursor to the beginning of the present sub-tree level:

$rs.C{:}per[x;l, u, r] = rs.C{:}per[l, u, x;r]$

$v = C{:}per[nil, u, r] \supset rs.v = v$

The *in[x]* operation for inserting an element before the cursor:

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

The *del* operation for deleting the element immediately after the cursor:

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

We can now proceed to the compilation issue. 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 on the form

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

where each of the A_i is a *specification clause* and has the form

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

where x and y are surrounding-valued *expressions*, and where the *literals* P_i are formed using the notation that has been 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 any constraints in order to guarantee in general that a specification is solvable; that has to be verified individually for each 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

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

where u^0 and u^n are now restricted to be ***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 approximates a simple, fairly conventional programming language (and more precisely, a single-assignment language), while at the same time being 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 succession of sub-languages (each sub-language being a first-order logic with certain syntactic restrictions).

6. Decomposition and modification functions.

In the sub-languages we need the following predicates:

on T $ispfx[t] \equiv (\exists\, x,y)\ t = x;y$

(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 $isper[p] \equiv (\exists\, l,u,r)\ p = per[l,u,r]$

(In other words, p is not •).

on U $nontop[C:p] \equiv isper[p]$

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

$J = s\text{-}l.h$

$x = s\text{-}tm.h$ (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 which 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

$e = s\text{-}e.a$

$m = s\text{-}m.a$

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 compositon 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 following 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 the name, the types of the domain and range, the precondition under which the function 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 -> L *lbl.J:p = J*	
pc	U -> P *pc.J:p = p*	
rg	U -> T *rg.J:per[l,u,r] = r*	*nontop[u] required for rg.u*
lf	U -> T *lf.J:per[l,u,r] = l*	*nontop[u] required for lf.u*
ow	U -> U *ow.J:per[l,u,r] = u*	*nontop[u] required for ow.u*
slf	T -> (U -> U) *slf[t].C:per[l,u,r] = C:per[t,u,r]*	*nontop[u] required for slf[t].u*
srg	T -> (U -> U) *srg[t].C:per[l,u,r] = C:per[l,u,t]*	*nontop[u] required for srg[t].u*
slbl	L -> (U -> U) *slbl[J].C:p = J:p*	
embed	LxTxT -> (U -> 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] ⊃ *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

$r \in \mathsf{T} \supset ispfx[r] \vee r = nil$

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

7. LPSL - Linear Program Sub-Language.

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

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

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

I. Simple constraints:

$m, n, p \geq 0$
$0 \leq h \leq m$

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

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

where *o* is an expression for a mapping U -> 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 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 fresh. 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 be arranged to 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, then 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, and the succession of such states that are obtained by successive update operations, are represented in our logic by a sequence of u_k variables. The actual implementation in a conventional programming language, can contain one single variable u, and do successive assignments to it, and/or successive side-effect operations on its value.

b) if P_i is of type (2) above, then 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 that had not previously been assigned.

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 be assigned before they are used.

When P_i is of type (3), i.e. a condition z, then it should be viewed as the condition in an *if* statement (of a conventional programming language); the subsequent P_{i+1}... constitute the *then* branch of the *if* statement, and 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 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 the type checks of programming languages.

e) $u^n \in c_p$

In order to make the condition (d) effectively decidable in general, it

would have to be strenghtened 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:

$$
\begin{aligned}
&(\forall\ v^1, u^0, u^1, u^2)\\
&\quad nontop[u^0] \wedge\\
&\quad v^1 = rg.u^0 \wedge\\
&\quad ispfx[v^1] \wedge\\
&\quad u^1 = slf[hd[v^1];(lf.u^0)].u^0 \wedge\\
&\quad u^2 = srg[tl[v^1]].u^1 \quad \supset\\
&\qquad nx.u^0 = u^2
\end{aligned}
$$

If an IMS theory is designed in the way that was suggested above, then that LPSL clause must be a consequence in the theory, from the following specification clause:

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

To verify that the (d) constraint 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 $slf[...].u^0$. The precondition $ispfx[v^1]$ legitimizes the use of $hd[v^1]$ and $tl[v^1]$. Finally, $u^1 = slf[...].u^0$ implies $nontop[u^1]$ (above) which legitimizes $u^2 = srg[...].u^1$.

The idea with the LPSL clause is that, besides being a consequence of a specification clause, it should express a procedure for executing the operation, in those cases accounted for by the specification clause. In other words, there should be a simple, essentially syntactic transformation which takes an LPSL clause into a *reasonable* procedure, in a conventional programming language, for doing the same thing. The intuitions of how the LPSL constructs are related to programming-language constructs, have already been given, and the constraints that have been set for LPSL clauses have been dictated by this goal. On the other hand, we are not yet ready to give a strict proof for this correspondence between LPSL and programming languages, and the set of constraints that have been given here should therefore be seen as provisional. In the simple cases that we use for examples in this paper, it should be fairly clear, anyway, 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 Sub-Language.

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

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

When these are transformed into LPSL, we often obtain LPSL clauses where the first few A_i are the same. It is natural to introduce a sub-language where they can be shared, and which then contains 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, defined by:

A *computation rule in BPSL* has the form

$P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_p \rightarrow u$

where $p \geq 0$, the successive P_i satisfy the same conditions as in LPSL, and u is either a variable u^m where $m \leq n$, or it is an expression

branch
if R_1
if R_2
...

where 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 a syntactic restriction on computation rules in BPSL which corresponds to the restriction on LPSL clauses, but in order to make it understandable, we must first make clear the meaning of the BPSL procedure.

First, as an example, the following is a BPSL procedure:

```
(∀ v¹, u⁰, u¹, u²) nx.u⁰ =/
        nontop[u⁰] ->
        v¹ = rg.u⁰ -> branch
              if ispfx[v¹] ->
                 u¹ = slf[hd[v¹];(lf.u⁰)].u⁰ ->
                 u² = srg[tl[v¹]].u¹ -> u²
              if v¹ = nil -> u⁰
```

and it 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; the second *if* clause the second line in the specification. Notice also that the second *if* clause uses u^0, not u^2, i.e. the increments of variable numbers proceed 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, then

$op.u =/ v$

has the same meaning as

$op.u = v$

The expression

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

has the same meaning as

$P_1 \wedge P_2 \wedge \ldots \wedge P_p \supset x =/ u$

The expression

x =/ branch
if u_1
if u_2
...

has the same meaning as

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

Using these rules, it is clearly possible to re-write each BPSL procedure as a conjunction of LPSL clauses (apart from the constraints). We shall refer to it as the ***flat form*** of the BPSL procedure. Our example above of the BPSL procedure for *nx* is then re-written 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 the natural constraint on BPSL procedures that each of the conjuncts in the flat form must satisfy the constraints on LPSL clauses (*).

If a BPSL procedure shall be a usable program, it must account for all cases that may occur in the IMS where it is used. The completeness criterium is that in any expression

$P_1 \rightarrow \ldots P_i \rightarrow$ *branch*
if $Q_1 \rightarrow v^1$
...
if $Q_n \rightarrow v^n$

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

$$P_1 \wedge \ldots \wedge P_i \supset (Q_1 \vee \ldots \vee Q_n)$$

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 condition *nontop*$[u^0]$ is required by the subsequent operations, and it means that the procedure is not complete in this sense. However, we notice that all IMS operations that were defined in section 5 assume the predicate

---

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

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

where $k \leq n$ without necessarily equality. Compare the definition of LPSL clauses above.

nontop of their argument, and also preserve that property. Therefore it would be reasonable to treat it as an invariant of the IMS, and to only require that BPSL procedures shall be well defined and complete relative to the invariant. (This relaxes requirement III.(d) in the definition of LPSL).

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]    ^

up.J:per[nil, K:nil, r] = J:per[nil, K:nil, r]    ^

up.J:per[x;l, u, r] = up.J:per[l, u, x;r]
```

and a derived expression in BPSL is:

```
(∀ v¹, v², u⁰, u¹, u², u³) up.u⁰ =/
    nontop[u⁰] -> branch
    if ispfx[lf.u⁰] ->
        u¹ = bk.u⁰ ->
        u² = up.u¹ -> u²
    if lf.u⁰ = nil ->
        v¹ = lbl.u⁰ ->
        v² = rg.u⁰ -> branch
        if nontop[ow.u⁰] ->
            u¹ = ow.u⁰ ->
            u² = slf[(lbl.u¹):v²;(lf.u¹)].u¹ ->
            u³ = slbl[v¹].u² -> u³
        if pc[u¹] = nil -> u⁰
```

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

10. IPSL - the Implicit-surrounding Program Sub-Language.

The LPSL and BPSL use explicit variables u^0, u^1, ... for the successive surroundings that are the states of the machine. When an expression in these sub-languages is transformed into a conventional program, as is readily done, it is of course possible to use one 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 sub-language, IPSL, where 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
$(\forall\ v^1,\ldots\ v^m,\ u^0,\ldots\ u^l)\ op[v^1,\ldots\ v^h].u^0 =/ R$
is transformed into
$Op[v^1,\ldots\ v^h] =/$ (local $v^{h+1},\ldots\ v^m$) R'

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

A computation rule in BPSL,

$P_1 \rightarrow \ldots P_p \rightarrow u$

is transformed into

$P'_1 \rightarrow \ldots P'_p$

if u is a single variable, and into

$P'_1 \rightarrow \ldots P'_p \rightarrow u'$

if u is a branch expression. In the latter case, the if clauses in the branch expression are each 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 $a.u^{k-1}$, is transformed into an expression

x'

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

$v^k = x$

is transformed to 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 can not be transformed.

Functions U -> U with an explicit variable as the argument are capitalized and the argument is omitted. Thus:

nontop[u] -> Nontop
lbl.u -> Lbl
pc.u -> Pc
rg.u -> Rg
lf.u -> Lf
ow.u -> Ow
slf[t].u -> Slf[t']
srg[t].u -> Srg[t']
slbl[J].u -> Slbl[J']
embed[J,l,r].u -> Embed[J',l',r']

To account for those cases where the argument to a function from U to U is not a variable symbol, we introduce the operator ⊛ defined by

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

so that e.g. *nontop[ow.u⁰]* can be rewritten as *Nontop⊛Ow.*

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

Finally, as a dose of syntactic sugaring, if the final step in a branch has the form

if Q -> u^m

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

if Q' -> Ident

rather than

if Q'

For example, the definition of *nx* that was given in BPSL in section 8, will be re-written into:

```
Nx =/ (local v1)
      Nontop ->
      v1 = Rg -> branch
            if ispfx[v1] ->
                  Slf[hd[v1];Lf] ->
                  Srg[tl[v1]]
            if v1 = nil -> Ident
```

In this way, we have moved our notation closer to what we find in a conventional programming language, but still only as a syntactic shift of notation. The semantics of first-order logic is retained.

11. Transformations to stack machine sub-languages.

The sub-languages that were 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. One use of this observation may be to write a translator from (e.g.) BPSL to a conventional programming language. Such a translator will just do a syntactic shift into a subset of the target language. We will then have arranged a transformation from a specification language to a programming language, from where it can then be compiled for execution on a conventional machine. These transformations are illustrated in figure 5.

We could however achive a 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 that 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 intutively corresponds to the machine language for a stack machine.

We define the following domains:

B = *{true, false}*
Has those two distinguished objects as members.

Q = **B** ∪ **M** ∪ **H** ∪ **T**
This is the domain of practically everything. It is needed for defining

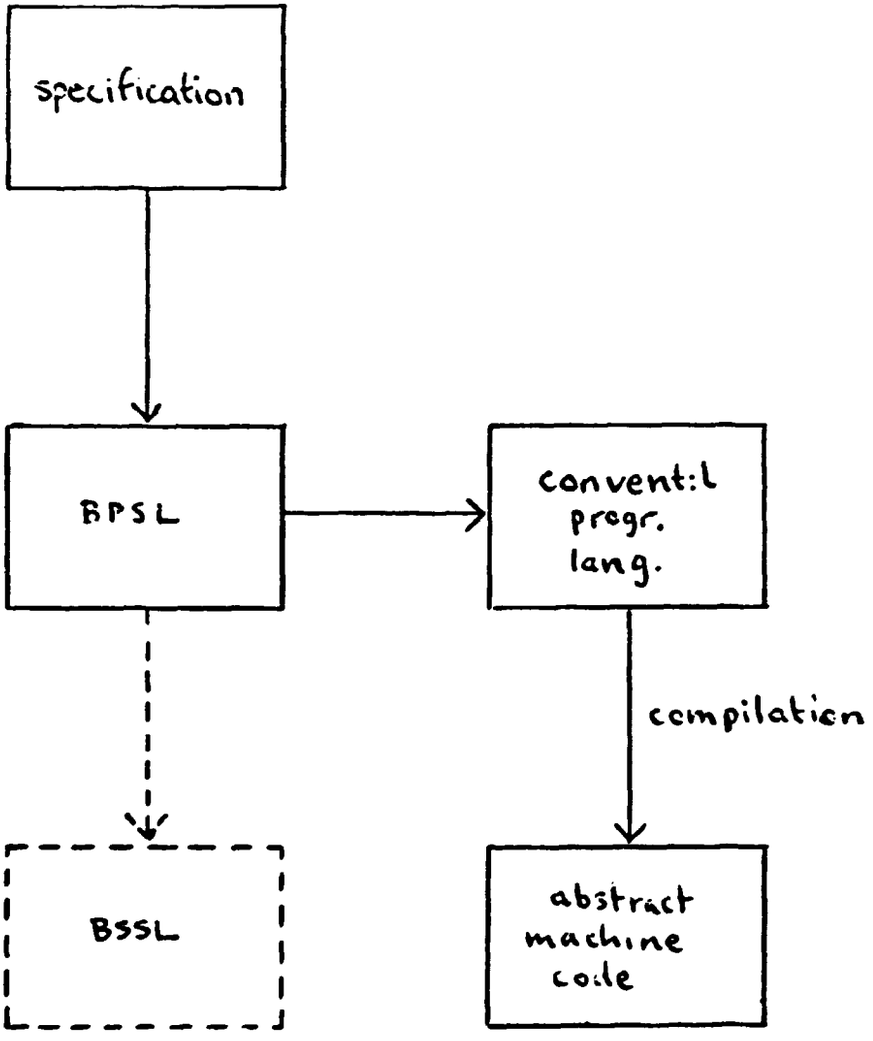

specification
BPSL
convent:l
progr.
lang.
compilation
BSSL
abstract
machine
code

Figure 5.

counterparts of registers and stacks.

S = U × Q* × Q

This 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.

We shall now define a sub-language, LSSL, which is analoguous to LPSL except that it uses states whenever LPSL uses surroundings, and 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 -> (S -> S)** **push[x].***<u,d,q>* = *<u, x;d, q>*
pop	**S -> S** **pop.***<u, x;d, q>* = *<u,d,x>*
reg	**S -> Q** **reg.***<u,d,q>* = *q*
test	predicate on **S**, defined as **test.***<u,d,q>* ≡ *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 those functions whose range is other than **U**, the counterpart pushes an element on the stack:

nontop	*nontop.u* ⊃	**nontop.***<u,d,q>* = *<u,* **true***;d, q>*
	¬ *nontop.u* ⊃	**nontop.***<u,d,q>* = *<u,* **false***;d, q>*

lbl	**lbl.***<u,d,q>* = *<u, (lbl.u);d, q>*
pc	**pc.***<u,d,q>* = *<u, (pc.u);d, q>*
rg	**rg.***<u,d,q>* = *<u, (rg.u);d, q>*
lf	**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 $\quad$ **ow.**$\langle u,d,q\rangle = \langle ow.u, d, q\rangle$

slf $\quad$ **slf.**$\langle u, x;d, q\rangle = \langle slf[x].u, d, q\rangle$

srg $\quad$ **srg.**$\langle u, x;d, q\rangle = \langle srg[x].u, d, q\rangle$

slbl $\quad$ **slbl.**$\langle u, x;d, q\rangle = \langle slbl[x].u, d, q\rangle$

embed $\quad$ **embed.**$\langle u, r;J;l;d, q\rangle = \langle embed[l,J,r].u, d, q\rangle$

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

pfx $\quad$ **pfx.**$\langle u, y;x;d, q\rangle = \langle u, (x;y);d, q\rangle$

hd $\quad$ **hd.**$\langle u, x;d, q\rangle = \langle u, (hd.x);d, q\rangle$

ispfx $\quad ispfx.x \supset$ **ispfx.**$\langle u, x;d, q\rangle = \langle u,$ **true**$;d, q\rangle$
and a similar definition for the case of
$\neg\ ixpfx.x$

equal $\quad x=y \supset$ **equal.**$\langle x, y;x;d, q\rangle = \langle u,$ **true**$;d, q\rangle$
and so forth.

Starting from an expression in LPSL, which represents operations on surroundings, we can now rewrite it as an expression in LSSL which represents 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

$s^2 =$ **push**$[v^1].s^1$ ->
$s^3 =$ **tl**$.s^2$ ->
$s^4 =$ **srg**$.s^3$

Here the first operation pushes the value of v^1 on the stack; the second operation does a *tl* operation on the top element of the stack, and the third operation 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, where the stack is first grown by operations such as **push** with a single variable parameter; **rg**, **lf**, etc., later modified by operations that take arguments and results on the stack, and finally depleted by

an operation such as **srg**.

type (2) literals in LPSL are similarly transformed to a sequence of literals in LSSL, where the last two clauses are:

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

type (3) literals, finally, are transformed to sequences of LSSL literals ending with

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

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

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

We can now see why the register component of the state is needed even for a stack machine: 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 that contains the popped stack, to an *s* variable. That is taken care of by the register as shown in the example. Since we do not do any operations on the contents of the register, and do not 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, resulting in a sub-language 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, and more precisely, and exit that re-sets 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 in *thin italic* at the end of each line. We then have something which feels like the machine language for a stack machine, except for how variables are handled:

```
Nx =/ (local v')
   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' = Reg -> branch     bind v' to y
   >> Push[v'] ->         push y on stack
      Ispfx ->            check that y is not nil
      Pop ->              pop result of test to register
      Test ->             exit if text fails
      Push[v'] ->         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 surr.ing
      Push[v'] ->         push y = x;r on top again
      Tl->                now r is on top of stack
      Srg                 assign r as new list of right sisters in surr.ing
                          Current state is result.
   >> Push[v'] ->         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.
```

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

12. Proposed continued work.

The present paper has been semiformal, and a next step should be to verify, for strict definitions of all languages involved, that 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, and that is in principle a reasonable constraint, but then 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 reportoire of operations.

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

- introduce a real register machine, and/or smoothen 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 doing them by predicate-logic variables
- As BSSL is defined, if the **test** statement fails during execution of an if clause, then there should be an exit from the if clause, and the next if clause should be entered with the machine in the same state as it was when the first if clause was entered. In an implementation, this would require a copy of the state to be saved, but it is intuitively clear that the machine has the same surrounding and stack when the exit is done as when the first if clause was entered. In this and other ways, one should arrange that the machine can behave more like a real machine, while retaining the close correspondence through all the levels from specification to executable machine language.

References.

BLI82 Andrzej Blikle: *Desophisticating Denotational Semantics.* Invited paper, to appear in the proceedings of the "IFIP" World Computer Congress, 1983.

BOE80 Barry E. Boehm: *Developing Small-scale Application Software Products.* In: S.H. Lavington (ed): Information Processing 83. North-Holland, 1980.

SAN82A Erik Sandewall: *An Approach to Information Management Systems.* Departmental report, Software Systems Research Center, Linköping University, Sweden, July 1982.

Research Reports published 1976-1983.

(Continued.)

LiTH-MAT-R-80-01	Johan Elfström, Jan Gillqvist, Hans Holmgren, Sture Hägglund, Olle Rosin, Ove Wigertz: A Customized Programming Environment for Patient Management Simulations. Also in *Proc. of the 3rd World Conf. on Medical Informatics*, Tokyo, 1980.
LiTH-MAT-R-80-08	Dan Strömberg, Peter Fritzon: Transfer of Programs from LISP to BCPL Environments: An Experiment. Revised version as "Transfer of Programs from Development to Runtime Environments" in *BIT, vol 20*, no 4, 1980.
LiTH-MAT-R-80-18	H. Jan Komorowski: QLOG - The Software for Prolog and the Logic Programming. Also in *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary, 1980.
LiTH-MAT-R-80-20	Pär Emanuelson, Anders Haraldsson: On Compiling Embedded Languages in Lisp. Also in *Proc. of the 1980 LISP Conf.*, Stanford, Calif, 1980.
LiTH-MAT-R-80-22	Erik Sandewall, Henrik Sörensen, Claes Strömberg: A System of Communicating Residential Environments. Also in *Proc. of the 1980 LISP Conf.*, Stanford, Calif, 1980
LiTH-MAT-R-80-23	Uwe Hein: Interruptions in Dialogue. Also in D. Metzing (ed), *Dialogmuster und Dialogprozesse*. Hamburg, Buske, 1981.
LiTH-MAT-R-80-24	Anders Haraldsson: Experiences from a Progam Manipulation System.
LiTH-MAT-R-80-27	Erik Tengvald: En Intuitiv Förklaring till Kildalls Algoritm
LiTH-MAT-R-80-28	Erik Tengvald: A Note Comparing Two Formalizations of Dataflow Algorithms.
LiTH-MAT-R-80-29	James W. Goodwin: Why Programming Environments Need Dynamic Data Abstractions. Also in *IEEE Trans. Software Eng., vol SE-7*, no 5, 1981.
LiTH-MAT-R-80-30	Lars Wikstrand, Sture Hägglund: A System for Program Analysis and its Application as a Tool for Software Development and Program Transfer.
LiTH-MAT-R-80-35	Erland Jungert: Deriving a Database Schema from an Application Model Based on User-defined Forms.
LiTH-MAT-R-80-36	James W. Goodwin and Uwe Hein: Artificial Intelligence and the Study of Language.
LiTH-MAT-R-80-37	Sture Hägglund: Towards Control Abstractions for Interactive Software. A Case Study.
LiTH-MAT-R-80-38	Sture Hägglund, Johan Elfström, Hans Holmgren, Olle Rosin, Ove Wigertz: Specifying Control and Data in the Design of Educational Software. Also in *Computers & Education, vol 6*, pp 67-76, 1982.
LiTH-MAT-R-80-39	Kenth Ericson, Hans Lunell: Redskap för kompilatorframställning
LiTH-MAT-R-80-41	Hans Lunell: Some notes on the terminology for Compiler-Writing Tools
LiTH-MAT-R-80-42	Östen Oskarsson, Henrik Sörensen: Integrating Documentation and Program Code
LiTH-MAT-R-81-01	Erik Sandewall, Claes Strömberg, Henrik Sörensen: Software Architecture Based on Communicating Residential Environments. Also in *Proc. of the 5th Int. Conf. on Software Engineering*, San Diego, 1981.
LiTH-MAT-R-81-02	H. Jan Komorowski, James W. Goodwin: Embedding Prolog in Lisp: An Example of a Lisp Craft Technique.
LiTH-MAT-R-81-03	Ola Strömfors, Lennart Jonesjö: The Implementation and Experiences of a Structure-Oriented Text Editor. Also in *Proc of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 8-10, (SIGPLAN NOTICES, vol 16, no 6) 1981.
LiTH-MAT-R-81-04	Sture Hägglund m fl: 80-talets elektroniska kontor: Erfarenheter från LOIS-projektet.
LiTH-MAT-R-81-05	Peter Fritzson: Distribuerad PATHCAL: Förslag till ett distribuerat interaktivt programmeringssystem för PASCAL.
LiTH-MAT-R-81-07	Alexander Ollongren: On the Implementation of Parts of Meta-IV in Lisp.
LiTH-MAT-R-81-10	Östen Oskarsson: Construction of Customized Programming Languages.
LiTH-MAT-R-81-19	Andrzej Blikle: Notes on the Mathematical Semantics of Programming Languages. (Lecture notes.)
LiTH-MAT-R-81-20	Erik Sandewall: Unified Dialogue Management in the Carousel System. Presented at the *ACM Conference on Principles of Programming Languages*, Albuquerque, NM, 1982. Appeared in print in *N. Naffah (ed.) Office Information Systems*, North Holland, 1982.

(Continued on next page.)

Research Reports published 1976-1983.

(Continued.)

LiTH-MAT-R-82-02 Uwe Hein: Constraints and Event Sequences.

LiTH-MAT-R-82-03 Pär Emanuelson: From Abstract Model to Efficient Compilation of Patterns. Also in *Proc. of the 5th Int. Conf. on Programming*, Turin, 1982. Revised version to appear in *Science of Computer Programming.*

LiTH-MAT-R-82-04 Uwe Hein: Kunskapsteori: Representation, Manipulation och Organisation av Kunskap - Del I - den teoretiska ramen. (Lecture notes).

LiTH-MAT-R-82-05 Hans Karlsson, Roland Lindvall, Olle Rosin, Erik Sandewall, Henrik Sörensen and Ove Wigertz: Experience from Computer Supported Prototyping for Information Flow in Hospitals. *Proc. of the ACM SIGSOFT Second Software Engineering Symposium: Workshop on Rapid Prototyping*, Columbia, Maryland, April 19-21, 1982.

LiTH-MAT-R-82-07 Uwe Hein: Kunskapsteori: Representation, Manipulation och Organisation av Kunskap - Del 2 - associativa nätverk. (Lecture notes).

LiTH-MAT-R-82-09 Hans Lunell: En konceptuell maskin för Pascal. (Preliminär version).

LiTH-MAT-R-82-10 Uwe Hein: Natural and Artificial Communications. - Some Reflections. To appear in: *Bolc, Hein and Hägglund (eds.) Models of dialog: theory and application. Munchen:W Carl Hanser.*

LiTH-MAT-R-82-15 Peter Fritzson: Fine-Grained Incremental Compilation for Pascal-Like Languages.

LiTH-MAT-R-82-16 Jan Maluszynski: Towards a Programming Language based on the Notion of Two-Level Grammar. (Revised version January 1983). To appear in *Theoretical Computer Science*, North-Holland.

LiTH-MAT-R-82-17 Erik Sandewall: Ny teknologi i kontorsdatasystem.

LiTH-MAT-R-82-18 Erik Sandewall, Sture Hägglund, Christian Gustafsson, Lennart Jonesjö, Ola Strömfors: Stepwise Structuring - A Style of Life for Flexible Software. Will be presented at the *1983 National Computer Conference*, Anaheim, Calif., May 1983.

LiTH-MAT-R-82-19 Erik Sandewall: An Approach to Information Management Systems.

LiTH-MAT-R-82-20 Erik Sandewall: An Environment for Development and Use of Executable Application Models. Presented at the *seminar "Software factory experiences"*, Capri, May 3-7, 1982.

LiTH-MAT-R-82-21 H. Jan Komorowski: An Abstract Prolog Machine. Also in *Proc. of the European Conf. on Integrated Interactive Computing Systems*, Stresa, 1982.

LiTH-MAT-R-82-23 James W. Goodwin: An Improved Algorithm for Non-monotonic Dependency Net Update.

LiTH-MAT-R-82-26 Jan Maluszynski, Jorgen Fischer Nilsson: Grammatical Unification.

LiTH-MAT-R-82-34 Dan Strömberg: Text Editing and Incremental Parsing.

LiTH-MAT-R-82-40 Sture Hägglund and Roland Tibell: Multi-Style Dialogues and Control Independence in Interactive Software. Presented at *the 1st European Conference on Cognitive Engineering*, Amsterdam, 1982.

LiTH-MAT-R-83-02 Erik Sandewall: Formal Specification and Implementation of Operations in Information Management Systems.To appear in: *Jan Heering and Paul Klint (eds.), Colloquium Programmeeromgevingen*, MC Syllabus, Mathematisch Centrum, Amsterdam 1983.