プログラミング言語 20－4
（1989. 2. 10）

記号計算抽象マシン

中村敦司　工学研究科
山田敏哉　情報学類
井田哲雄　電子・情報工学系
筑波大学

ASMと名付けた記号計算のための抽象マシンの形式的記述を与える。ASMは我々が現在開発しているメタ計算環境の核となる抽象マシンであり、現在はLispとPrologの処理系がその上で作成されている。形式的記述は、実現している処理系を正確、簡潔に表現できる。意味論的に全く異なるかのようにみえるLispとPrologのシステムが、同じ仮想マシンの上で資源を共有し、関数と述語を密接に結合するような実現が可能であることが明らかにされる。またこの記述により、我々の仮定しているアーキテクチャを形式的に論じることができるようになる。我々が形式的記述に用いた言語は、宣言的意味論に基づいている。

Abstract Symbolic Machine

Atsushi Nakamura
Doctoral Program
in Engineering

Toshiya Yamada
College of Information Sciences

Tetsuo Ida
Institute of Information Sciences
and Electronics

University of Tsukuba
1-1-1 Tennoudai Tsukuba Ibaraki Japan 305

We give a formal description of the Abstract Symbolic Machine called ASM. ASM is the basis of the meta computing environment which we are developing. So far Lisp and Prolog systems are constructed on ASM. The formal description is intended to precisely and concisely describe the implemented systems, and to show how Prolog and Lisp which at first sight look quite different from semantical point of view can be implemented on the common machine sharing most resources and enabling fine grain communication between functions of Lisp and predicates of Prolog. The description enables us to reason about architectures formally. The language we use for formal description is based on denotational semantics.

1. Introduction

In this paper we are concerned with a formal description of an Abstract Symbolic Machine (to be called ASM for short) which is the basis of the meta computing environment MC [1]. We have implemented a subset of COMMON LISP (both compiler and interpreter) and Prolog (interpreter) built on ASM. The purpose of the formal description of ASM is

(i) to precisely describe the implementation methods for Lisp and Prolog incorporated in MC, using concise mathematical notations,

and

(ii) to show how both Lisp and Prolog can be combined at an abstract machine level sharing common resources.

Since the full description of ASM is beyond the scope of this paper, we concentrate on the descriptions of formal semantics of ASM instructions using the modified denotational semantics.

As we give the formal semantics of ASM, we introduce many notations. In our view the notation is closely related to the fundamental concepts of underlying computational models. Hence, we hope that readers follow as far as possible the expressions which look like mathematical formulas.

2. Definition of Abstract Machines

We use following basic terminology in the language of formal description.

2.1. Definition symbol and string

(i) A word is a unit of expressions which is treated as a single denotation in the model of the machine.

(ii) A symbol is either a single word or a list of words.

(iii) A string is a sequence of symbols.

(iv) The length of a string is the number of symbols which constitutes the string.

(v) $[\alpha, \ldots, \beta]$ denotes a coerced symbol from strings $\alpha, \ldots, \beta$. In other words, it is treated as a single word rather than a string. We assume that 'reverse-coercion' is possible so that from $[\alpha, \ldots, \beta]$ we can extract each of $\alpha, \ldots, \beta$.

2.2. Definition machine

Let **I** be a program, Σ be a set of states and S be a state transition map; $\Sigma \to \Sigma$. A machine is defined as a triplet $< \mathbf{I}, \Sigma, S >$.

A program **I** is defined by instruction string $\mathbf{I} = i_0 i_1 \ldots i_n$, where i_j is an instruction symbol. i_0 is considered as an initial instruction. Each instruction symbol i_j is associated with an integer j called label.

An architecture of a machine is specified by Σ and S. Since ASM can be viewed as a combined Lisp and Prolog abstract machine, we give specifications of two machines separately and later we show how the two are combined. First for each Lisp and Prolog abstract machines we give Σ and S.

3. Lisp Abstract Machine

Lisp abstract machine $\mathcal{M}_\mathcal{L}$ is defined as $< \mathbf{I}, \Sigma_\mathcal{L}, S_\mathcal{L} >$.

3.1 Definition state of $\mathcal{M}_\mathcal{L}$

A state of the abstract machine $\mathcal{M}_\mathcal{L}$ is a snapshot of stores which are defined as 7-tuple

$$< \mathfrak{K}, \mathfrak{R}, \mathfrak{M}^*, \mathfrak{C}^*, {}^*\mathfrak{F}, \mathfrak{W}^*, {}^*\mathfrak{V} >$$

where $\mathfrak{K}$ is an instruction store,
$\mathfrak{R}$ is a register,
$\mathfrak{M}$ is a multiple-value stack,
$\mathfrak{C}$ is a control stack,
$\mathfrak{F}$ is a frame stack,
$\mathfrak{W}$ is a full-word area, and
$\mathfrak{V}$ is a cons area.

The 7-tuple is called configuration.

Abusing the above notation, we write $\kappa \in \mathfrak{K}$ etc., to denote a string κ stored in $\mathfrak{K}$. With this notation we let $\Sigma_\mathcal{L} = < \mathfrak{K}, \mathfrak{R}, \mathfrak{M}, \mathfrak{C}, \mathfrak{F}, \mathfrak{W}, \mathfrak{V} >$ and define a state of machine $\mathcal{M}_\mathcal{L}$ as 7-tuple

$$< \kappa, r, \mu, \theta, \eta, \omega, \lambda > \in \Sigma_\mathcal{L},$$

where $\kappa \in \mathfrak{K}$,
$r \in \mathfrak{R}$,
$\mu \in \mathfrak{M}$,
$\theta \in \mathfrak{C}$,
$\eta \in \mathfrak{F}$,
$\omega \in \mathfrak{W}$ and
$\lambda \in \mathfrak{V}$.

The superscript * in $\mathfrak{H}^*$ or ${}^*\mathfrak{H}$ for $\mathfrak{H} = \mathfrak{M}, \mathfrak{C}, \mathfrak{F}, \mathfrak{W}, \mathfrak{V}$ expresses a property that a string stored in $\mathfrak{H}^*$ is concatenated only from right, and ${}^*\mathfrak{H}$ from

本研究は、文部省科学技術研究費補助金一般研究(C)62580030 および、昭和63年度筑波大学学内プロジェクト研究費の補助を得て行なわれた.

left. Here $*\mathsf{H}$ or $\mathsf{H}*$ is conceptually regarded as a stack with read-write head position denoted as *.

3.2 Notations

1. Lower-case Greek alphabets denote strings and Roman alphabets denote symbols.
2. ε denotes null string.
3. - denotes some string or symbol stored in stores.
4. A lower-case Greek letter α with a superscript n, i.e. α^n, denotes a string α whose length is n.
5. ~ with a superscript n, i.e. $\sim^n$, denotes a string of length n. Symbols which constitute that string are unspecified.

3.3 Definition state transition map S_L

The state transition map $S_{\mathcal{L}}$: $\mathit{instructions} \times \Sigma \rightarrow \Sigma$ is defined by exhaustive enumeration of state transitions for each instruction of $\mathcal{M}_{\mathcal{L}}$.

Although the number of instructions of the Lisp abstract machine is small, it is beyond the scope of this paper to give a full description of $S_{\mathcal{L}}$. We only give partial description of $S_{\mathcal{L}}$ for representative instructions.

In the following description we use :

(1) a map φ_I which maps a label to a substring of program **I** is defined as follows;

$$\varphi_I : \mathit{labels} \rightarrow \mathit{instruction\ strings}$$

$$\varphi_I(j) = i_j\, i_{j+1} \ldots i_n , \quad 0 \leq j \leq n$$
$$\varphi_I(j) = \perp \qquad \text{otherwise}$$

Since **I** is implicitly given when talking about a machine $< \mathbf{I} , \Sigma_{\mathcal{L}} , S_{\mathcal{L}} >$ we generally omit subscript **I** of φ_I and write it simply as φ.

(2) Following maps which associate a symbol with values:

$$\mathit{update}_J : \mathbb{W} \times \mathbf{SYMB} \times \mathbf{U} \rightarrow \mathbb{W}$$
$$\mathit{lookup}_J : \mathbb{W} \times \mathbf{SYMB} \rightarrow \mathbf{U}$$

where **U** is a set of symbols as defined in 2.1(ii), and **SYMB** is a set of LISP SYMBOLs stored in $\mathbb{W}$.

update_J and lookup_J are a family of functions indexed by $J \in \{ \mathcal{C}, \mathcal{F}, \mathcal{P} \}$.

update_J (ω, y ,u) updates the value associated with SYMBOL y and J in ω with value u, and returns updated ω.

lookup_J (ω, y) searches for SYMBOL y in ω and returns the value associated with y and J.

The uniqueness of symbol y in ω is guaranteed by the system (which we do not discuss in this paper).

It is easy to see that

$$u = \mathit{lookup}_J (\mathit{update}_J (\omega, y ,u), y) .$$

Here are descriptions of $S_{\mathcal{L}}$. Table 1 gives intuitive meanings of each instruction.

(1) Data movement

$$S_{\mathcal{L}}\ (\mathsf{L} \quad \mathsf{R} \quad \mathsf{m})\ < - , - , - , - , \eta , - , - >$$
$$= \ < - , x , - , - , \eta , - , - >$$
where $\eta = [\, \tau^m\, x\, \tau' \,]\, \eta'$

note: $[\, \tau^m\, x\, \tau' \,]$ is a current frame

$$S_{\mathcal{L}}\ (\mathsf{ST} \quad \mathsf{R} \quad \mathsf{m})\ < - , v , - , - , [\, \tau^m\, x\, \tau' \,]\, \eta' , - , - >$$
$$= \ < - , v , - , - , [\, \tau^m\, v\, \tau' \,]\, \eta' , - , - >$$

(2) Control

$$S_{\mathcal{L}}\ (\mathsf{B} \quad \mathsf{l})\ < \kappa , - , - , - , - , - , - >$$
$$= \ < \kappa' , - , - , - , - , - , - > \quad \text{where } \kappa' = \varphi(\mathsf{l})$$

$$S_{\mathcal{L}}\ (\mathsf{ENTRY}\ \mathsf{n})\ < - , - , - , - , [\, \tau \,]\, \eta , - , - >$$
$$= \ < - , - , - , - , [\, \sim^n \tau \,]\, \eta , - , - >$$

$$S_{\mathcal{L}}\ (\mathsf{CALL} \quad \mathsf{id} \quad \mathsf{m} \quad \mathsf{n})\ < \kappa , - , - , - , \eta , \omega , - >$$
$$= < \kappa' , - , - , - , [\, [\, \kappa , \eta \,]\, \tau'^n \,]\ [\, \tau'' \,]\, \eta' , - , - >$$
where $\eta = [\, \tau^m\, \tau'^n\, \tau'' \,]\, \eta'$

and $\kappa' = \varphi(\mathit{lookup}_{\mathcal{F}}(\omega , \mathsf{id}))$

$$S_{\mathcal{L}}\ (\mathsf{RTN} \quad \mathsf{n})$$
$$< - , - , - , - ,[\, \tau^n\ [\, \kappa' , \eta' \,]\, \tau' \,]\, \eta , - , - >$$
$$= \ < -' , - , - , - , \eta' , - , - >$$

$$S_{\mathcal{L}}\ (\mathsf{DEALLOC} \quad \mathsf{n})\ < - , - , - , - , [\, \nu^n\, \tau \,]\, \eta , - , - >$$
$$= \ < - , - , - , - , [\, \tau \,]\, \eta , - , - >$$

$$S_{\mathcal{L}}\ (\mathsf{THROW} \quad \mathsf{m})\ \ < \kappa , - , - , \theta , [\, \tau^m\, x\, \tau' \,]\, \eta , - , - >$$
$$= \ < \kappa' , - , - , \theta' , \eta' , - , - >$$
where $[\, \theta' [\, t , \eta' , 1 \,]\,] = \mathit{lookup}_c(x)$, $t = \mathit{Catch}$

and $\kappa' = \varphi(1)$

$$S_{\mathcal{L}}\ (\mathsf{RETURN} \quad \mathsf{m})\ \ < \kappa , - , - , \theta , [\, \tau^m\, x\, \tau' \,]\, \eta , - , - >$$
$$= \ < \kappa' , - , - , \theta' , \eta' , - , - >$$
where $x = [\, \theta' [\, t , \eta' , 1 \,]\,]$, $t = \mathit{Block}$

and $\kappa' = \varphi(1)$

$$S_{\mathcal{L}}\ (\mathsf{GO} \quad \mathsf{m})\ < \kappa , - , - , \theta , [\, \tau^m\, x\, \tau' \,]\, \eta , - , - >$$
$$= \ < \kappa' , - , - , \theta' , \eta' , - , - >$$
where $x = [\, \theta' [\, t , \eta' , 1 \,]\,]$, $t = \mathit{Go}$

and $\kappa' = \varphi(l)$

$$S_{\mathcal{L}}\ (\text{PUSHCONT } t\ l)\ <-,-,-,\theta,\eta,-,->$$
$$= \ <-,-,-,\theta[t,\eta,l],\eta,-,->$$

$$S_{\mathcal{L}}\ (\text{UNBINDPOP})\ <-,-,-,\theta[t,x,a],-,\omega,->$$
$$= \ <-,-,-,\theta,-,\omega',->$$
where $\omega' = \mathit{update}_c(\omega, x, a)$
and $t = \mathit{Bind}$

$$S_{\mathcal{L}}\ (\text{CPOP } n)\ <-,-,-,\theta\sim^{n},-,-,->$$
$$= \ <-,-,-,\theta,-,-,->$$

(3) List processing

$$S_{\mathcal{L}}\ (\text{CAR } m\ l)\ <\kappa,-,-,-,\eta,-,\lambda>$$
$$= \begin{cases} <\kappa,v,-,-,\eta,-,\lambda> & \text{if } x \in \mathbf{U} \text{ and } x = [[v,v']\lambda] \\ <\kappa',-,-,-,\eta,-,\lambda> & \text{otherwise} \end{cases}$$
where $\eta = [\tau^m\ x\ \tau']\ \eta'$ and $\kappa' = \varphi(l)$

$$S_{\mathcal{L}}\ (\text{CDR } m\ l)\ <\kappa,-,-,-,\eta,-,\lambda>$$
$$= \begin{cases} <\kappa,v',-,-,\eta,-,\lambda> & \text{if } x \in \mathbf{U} \text{ and } x = [[v,v']\lambda] \\ <\kappa',-,-,-,\eta,-,\lambda> & \text{otherwise} \end{cases}$$
where $\eta = [\tau^{m-1}\ x\ \tau']\ \eta'$ and $\kappa' = \varphi(l)$

note : cons are defined by the following function

$$\text{cons } a\ b\ <\kappa,-,-,-,-,\omega,\lambda>$$
$$= \begin{cases} <\kappa,[\lambda'],-,-,-,\omega,\lambda'> & \text{if } \|\omega\ \lambda'\| < \text{heap-size} \\ \perp & \text{otherwise} \end{cases}$$
where $\lambda' = [a,b]\lambda$

(4) Type check

$$S_{\mathcal{L}}\ (\text{ERROR\_CAR } l)\ <\kappa,r,-,-,-,-,->$$
$$= \begin{cases} <\kappa',r,-,-,-,-,-> & \text{if } r = \text{Nil} \\ \perp & \text{otherwise} \end{cases}$$
where $\kappa' = \varphi(l)$

(5) Special variable binding

$$S_{\mathcal{L}}\ (\text{BINDPUSH } m)\ <-,-,-,\theta,\eta,\omega,->$$
$$= \ <-,-,-,\theta[t,x,a],\eta,\omega,->$$
where $\eta = [\tau^m\ x\ \tau']$, $t = \mathit{Bind}$
and $a = \mathit{lookup}_c(\omega, x)$

(6) Multiple-values

$$S_{\mathcal{L}}\ (\text{SAVE\_MV } n\ m)\ <-,-,\mu,-,\eta,-,->$$
$$= \ <-,-,\mu,-,[\tau^m\ v^n\ \tau'']\ \eta',-,->$$
where $\eta = [\tau^m\ \tau'^n\ \tau'']\ \eta'$ and $\mu = v^n\ \mu'$

$$S_{\mathcal{L}}\ (\text{RESTORE\_MV } n\ m)$$
$$<-,-,-,-,[\tau^m\ v^n\ \tau'']\ \eta',-,->$$
$$= \ <-,-,v^n,-,\eta,-,->$$
where $\eta = [\tau^m\ \tau'^n\ \tau'']\ \eta'$

4. Operation of a machine

The operation of a machine is explained via reduction. Below we define reduction for each Lisp and Prolog machine, although the notion of reduction can be generalized easily.

4.1. Definition reduction

Let $\sigma, \sigma' \in \Sigma_{\mathcal{L}}$.
Reduction $\rightarrow_{\mathcal{M}_{\mathcal{L}}}$ is a binary relation on states defined by the following :

$$\forall \sigma, \sigma' \in \Sigma_{\mathcal{L}},$$
$$\sigma = <\kappa, r, \mu, \theta, \eta, \omega, \lambda>,$$
$$\sigma' = <\kappa', r', \mu', \theta', \eta', \omega', \lambda'>$$

$$\sigma \rightarrow_{\mathcal{M}_{\mathcal{L}}} \sigma' \quad \Leftrightarrow \quad S_{\mathcal{L}}\, i\ \ \sigma = \sigma'$$

Intuitionally, instruction i is fetched from instruction store and executed by $S_{\mathcal{L}}$ and the result is a new state σ'. Reduction for $\mathcal{M}_{\mathcal{P}}$ is defined similarly. A machine operation is a transitive closure $\rightarrow^*_{\mathcal{M}}$ of $\rightarrow_{\mathcal{M}}$. A machine is said to be halt when $S_{\mathcal{L}}\, i\ \ \sigma = \sigma'$, where
$\sigma' = <\varepsilon, r', \mu', \theta', \eta', \omega', \lambda'>$.
In other words, a machine is halt when there is no more to be executed. An answer for the execution of program **I** may then be extracted from stores.
A state
$\sigma' = <\perp, r', \mu', \theta', \eta', \omega', \lambda'> = \perp$
denotes error, which we do not further elaborate.

5. Prolog Abstract Machine definition

As in the case of Lisp Abstract machine $\mathcal{M}_{\mathcal{L}}$, Prolog Abstract machine $\mathcal{M}_{\mathcal{P}}$ is defined as $<$ **I** , $\Sigma_{\mathcal{P}}, S_{\mathcal{P}}>$.

5.1. Definition state of $\mathcal{M}_{\mathcal{P}}$

A state of the abstract machine $\mathcal{M}_{\mathcal{P}}$ is a snapshot of stores defined as

$$< \mathsf{K}, [\mathsf{A}^*, {}^*\mathsf{H}, \mathsf{L}, \mathsf{B}], \mathsf{C}^*, {}^*\mathsf{F}, \mathsf{W}^*, {}^*\mathsf{U} >$$

where **L** is a list pointer,
B is a backtrack pointer,
A is an argument stack,
H is a temporary stack, and
other symbols have the same meaning as in $\mathcal{M}_{\mathcal{L}}$.
This architecture is based on WAM [2].

5.2. Notations

Following additional notations are used in describing $\mathcal{M}_{\mathcal{P}}$.

1. $\mu \in \mathbf{A}$, $\xi \in \mathbf{H}$
2. $\rightarrow$ is a special marker, which denotes a special point in **F**.
3. A backquoted α, i.e. $`\alpha$ denotes a string that may contain a $\rightarrow$.
   Namely, $`\alpha \equiv \beta \rightarrow \gamma$
   or $\beta\gamma$

5.3. Definition state transition map $S_{\mathcal{P}}$

Following auxiliary functions are used in the definition of $S_{\mathcal{P}}$.

(1) *deref*: **Term** × **F** × **U** → **TAG** × **Term**

where **TAG** = { list , atom ,
var-in-current-frame,
var-in-other-frame,
var-in-heap } ,

Term denotes a set of atoms, lists and variables , and
Uar denotes a set of variables.

deref(τ , η , λ) dereferences the first argument τ and returns a pair <t , v> of tag t and the dereferenced value v.
The kind of values returned depends on the tag and is classified as follows:

list	: cons cell other than variables
atom	:ground term other than lists
others	:uninstantiated variable

The tags in the last case indicate the place of this uninstantiated variable.

(2) *bind*: **Uar** × **Term** × **F** × **U** → **F** × **U**

bind(τ , v , η , λ) updates the variable τ with v. η or λ must reflect the change caused by the update, depending on where (in either **F** or **U**) v is allocated. The result of *bind*(τ , v , η , λ) is < η' , λ' >. One of which is the same as before.

(3) *unify*: **Term** × **Term** × **F** × **U**
→ { success , failure } × **F** × **U**

unify(τ , τ' , η , λ) unifies terms τ and τ'. It returns success or failure and updated η and λ , i.e. <x η' , λ'>. In the case of x = success, some of the variable may be instantiated. This change is reflected in η' and λ'

(4) *trail*: **C** × **Uar** × **F** × **B** → **C** is defined as follows:

$$trail(\theta , \tau^k , \eta^m , \eta' \rightarrow \eta''^n) = \begin{cases} \theta & \text{when } \tau^k \in \mathbf{F} \text{ and } n < k \leq m \\ \theta [\tau^k] & \text{otherwise} \end{cases}$$

(5) *mkvariable* : **U** → **U**

mkvariable(λ) allocates an unbound variable in heap. It returns updated λ , i.e. [*Unbound* , *Variable*] λ .

(6) *mkcons* : **U** × **Term** × **Term** → **U**

mkcons(λ , α , β) allocates a new cons cell in heap. It returns updated λ , i.e. [α , β] λ .

(7) *framevar* : **N** × **F** → **F**

where **N** is a set of integer number.

framevar(n , $`\eta$) returns a n-th variable allocated in current stack frame, i.e. [ $\tau$ , - , - , - ] where $`\eta = \eta'' \rightarrow [\tau'^n \tau , - , - , -] \eta'$.

Table 2 gives intuitive meanings of each instruction.

(1) Control

$S_{\mathcal{P}}$ (ENTRY n) $< - , - , - , \rightarrow [\tau] \eta , - , - >$
$= < - , - , - , \rightarrow [\sim^n , \varepsilon , \varepsilon , \tau] \eta , - , - >$

$S_{\mathcal{P}}$ (CALL id) $< \kappa , - , - , \eta \rightarrow \eta'^n , \omega , - >$
$= < \kappa' , - , - , \rightarrow [[\kappa , n]] \eta \eta' , \omega , - >$
where $\kappa' = \varphi(lookup_{\mathcal{P}}(\omega , id))$

$S_{\mathcal{P}}$ (EXECUTE id) $< \kappa , - , - , \eta \rightarrow \eta' , - , - >$
$= < \kappa' , - , - , \rightarrow [r] \eta \ \eta' , - , - >$
where $\eta' = [- , - , - , r] \eta''$
and $\kappa' = \varphi(lookup_{\mathcal{P}}(\omega , id))$

$S_{\mathcal{P}}$ (GB id) $< \kappa , - , - , \rightarrow [- , - , - , r] \eta , - , - >$
$= < \kappa' , - , - , \rightarrow [r] \eta , - , - >$
where $\kappa' = \varphi(lookup_{\mathcal{P}}(\omega , id))$

$S_{\mathcal{P}}$ (RTN) $< \kappa , - , - , \eta \rightarrow f \eta' \eta''^n , - , - >$
$= < \kappa' , - , - , \eta f \eta' \rightarrow \eta''^n , - , - >$
where $f = [- , - , - , [\kappa' , n]]$

$S_{\mathcal{P}}$ (DEALLOC n)
$< - , - , - , \rightarrow [\nu^n \tau , - , - , -] \eta , - , - >$

$$= \langle -, -, -, \rightarrow [\tau, -, -, -]\eta, -, - \rangle$$

$S_{\mathcal{P}}$ (CDEALLOC n) $\langle -, -, -, \grave{\eta}, -, - \rangle$

$$= \begin{cases} \langle -, -, -, \rightarrow [\tau, -, -, -]\eta'', -, - \rangle & \\ \qquad \text{if } \grave{\eta} = \rightarrow [\nu^n \tau, -, -]\eta'' & \\ \langle -, -, -, \grave{\eta}, -, - \rangle & \\ \qquad \text{otherwise} & \end{cases}$$

$S_{\mathcal{P}}$ (B l) $\langle \kappa, -, -, -, -, - \rangle$

$$= \langle \kappa', -, -, -, -, - \rangle \text{ where } \kappa' = \varphi(l)$$

(2) Clause group

$S_{\mathcal{P}}$ (TRY_ME_ELSE l n)

$$\langle -, [\mu, -, -, \grave{\eta}'], \theta, \rightarrow [-, \varepsilon, \varepsilon, -]\eta, -, - \rangle$$
$$= \langle -, [\mu, -, -, \grave{\eta}''], \theta, \grave{\eta}'', -, - \rangle$$

where $\grave{\eta}'' = \rightarrow [-, [\theta, \lambda, \grave{\eta}', \kappa', n], \mu, -]\eta$
and $\kappa' = \varphi(l)$

$S_{\mathcal{P}}$ (TRY l n)

$$\langle \kappa, [\mu, -, -, \grave{\eta}'], \theta, \rightarrow [-, \varepsilon, \varepsilon, -]\eta, -, \lambda \rangle$$
$$= \langle \kappa', [\mu, -, -, \grave{\eta}''], \theta, \grave{\eta}'', -, \lambda \rangle$$

where $\grave{\eta}'' = \rightarrow [-, [\theta, \lambda, \grave{\eta}', \kappa, n], \mu, -]\eta$
and $\kappa' = \varphi(l)$

$S_{\mathcal{P}}$ (TRY_ME_ONLY l)

$$\langle \kappa, -, -, \rightarrow [-, \varepsilon, \varepsilon, -]\eta, -, - \rangle$$
$$= \langle \kappa', -, -, \rightarrow [-, -, -, -]\eta, -, - \rangle$$

where $\kappa' = \varphi(l)$

note : Actually this instruction initiate two ε for garbage collector.

$S_{\mathcal{P}}$ (RETRY_ME_ELSE l n)

$$\langle -, -, -, \rightarrow [-, [-, -, -, -, -], -]\eta, -, - \rangle$$
$$= \langle -, -, -, \rightarrow [-, [-, -, -, \kappa, n], -]\eta, -, - \rangle$$

where $\kappa = \varphi(l)$

$S_{\mathcal{P}}$ (RETRY l n)

$$\langle \kappa, -, -, \rightarrow [-, [-, -, -, -, -], -]\eta, -, - \rangle$$
$$= \langle \kappa', -, -, \rightarrow [-, [-, -, -, \kappa, n], -]\eta, -, - \rangle$$

where $\kappa' = \varphi(l)$

$S_{\mathcal{P}}$ (TRUST_ME) $\langle -, [-, -, -, \grave{\eta}'], -, \grave{\eta}, -, - \rangle$

where $\grave{\eta} = \rightarrow [-, [-, -, \grave{\tau}, -, -], -, -]\eta''$

$$= \langle -, [-, -, -, \grave{\tau}], -, \grave{\eta}, -, - \rangle$$

note : backtrack is defined by the following function

$$\text{backtrack } \langle -, [-, -, -, \grave{\eta}], \theta, -, \omega, \lambda \rangle$$
$$= \langle \kappa, [\mu^n, -, -, \grave{\eta}''], \theta', \grave{\eta}'', \omega, \lambda \rangle$$

where $\eta = \eta' \rightarrow [-, [\theta', \lambda', \grave{\eta}'', \kappa, n], \mu^n\mu', -]$

(3) Indexing

$S_{\mathcal{P}}$ (SWITCH_ON_TERM s lv lc ll)

$$\langle \kappa, [\mu, -, -, -], -, \grave{\eta}, -, \lambda \rangle$$

when $s = A_n$ (other cases not given in this paper)

$$= \langle \kappa', [\mu, -, \delta, -], -, \grave{\eta}, -, \lambda \rangle$$

where $\mu = \mu'^n [\tau] \mu''$, $\grave{\eta} = \eta' \rightarrow \eta''$,
$\langle t, \delta \rangle = \mathit{deref}(\tau, \eta'', \lambda)$, and

$$\kappa' = \begin{cases} \varphi(lc) & \text{when } t = \text{atom} \\ \varphi(ll) & \text{when } t = \text{list} \\ \varphi(lv) & \text{otherwise } (\delta \in \mathbf{Var}) \end{cases}$$

$S_{\mathcal{P}}$ (SWITCH_ON_CONSTANT table l)

$$\langle \kappa, [-, -, x, -], -, -, -, - \rangle$$
$$= \langle \kappa', [-, -, x, -], -, -, -, - \rangle$$

where table is a map which maps an atom name to a label or ε.

$$\kappa' = \begin{cases} \varphi(\text{table}(x)) & \text{if } x \in \mathbf{SYMB} \text{ and table}(x) \neq \varepsilon \\ \varphi(l) & \text{otherwise} \end{cases}$$

(4) List pointer movement

$S_{\mathcal{P}}$ (CONS_SAVE X_n)

$$\langle \kappa, [-, \xi x \xi^n, \tau, -], -, -, -, - \rangle$$
$$= \langle \kappa, [-, \xi [\tau] \xi^n, \tau, -], -, -, -, - \rangle$$

$S_{\mathcal{P}}$ (CONS_RESTORE X_n)

$$\langle \kappa, [-, \xi [\tau] \xi^n, -, -], -, -, -, - \rangle$$
$$= \langle \kappa, [-, \xi [\tau] \xi^n, \tau, -], -, -, -, - \rangle$$

(5) Put operation

$S_{\mathcal{P}}$ (PUT_VARIABLE s ac)

$$\langle -, [\mu^n x \mu', \xi, -, -], -, \grave{\eta}, -, \lambda \rangle$$

when $ac = A_n$ (other cases not given in this paper)

$$= \begin{cases} \langle -, [\mu^n [\tau] \mu', \xi, -, -], -, \nu \rightarrow \eta'', -, \lambda \rangle & \\ \qquad \text{when } s = Y_m & \\ \qquad \text{where } \tau = \mathit{framevar}(m, \grave{\eta}) & \\ \qquad \qquad \grave{\eta} = \nu \rightarrow \eta' & \\ \qquad \text{and } \langle \eta'', - \rangle = \mathit{bind}(\tau, \mathit{Unbound}, \eta', -) & \\ \langle -, [\mu^n [\lambda'] \mu', \xi', -, -], -, \grave{\eta}, -, \lambda' \rangle & \\ \qquad \text{when } s = X_m & \\ \qquad \text{where } \xi = \nu' y \nu''^m, \ \xi' = \nu' [\lambda'] \nu''^m & \\ \qquad \text{and } \lambda' = \mathit{mkvariable}(\lambda) & \end{cases}$$

$S_{\mathcal{P}}$ (PUT_VALUE s ac)

$$\langle -, [-, -, \lambda^n, -], -, \grave{\eta}, -, \lambda' [\text{Nil}, -] \lambda''^{n-1} \rangle$$

When $s = Y_m$ and $ac = \text{car}$
(other cases not given in this paper)

$$= \; < - , [- , - , \lambda^{n} , -] , - , `\eta , - , \lambda'[a, -] \lambda''^{n-1} >$$
$$\text{where } `\eta = \tau \rightarrow [\nu^{m}\, a\, \nu' , - , - , -]\, \tau'$$

$S_{\mathcal{P}}$ (PUT_UNSAFE_VALUE Y_n ac)

$$< - , [\mu^{m}\, x\, \mu' , - , - , -] , - , `\eta , - , \lambda >$$
$$\text{where } `\eta = \tau \rightarrow \eta' , \eta' = [\nu^{n} [\tau']\, \nu' , - , - , -]\, \zeta'$$
$$\text{and } <t , \delta> = \mathit{deref}(\tau' , \eta' , \lambda)$$

when ac = A_n (other cases not given in this paper)

$$= \begin{cases} < - , [\mu^{m} [\lambda'] \mu' , - , - , -] , - , \rightarrow \eta'' , - , \lambda' > \\ \quad \text{if } \tau = \varepsilon \text{ and } t = \text{var-in-current-frame} \\ \quad \text{where } \lambda' = \mathit{mkvariable}(\lambda) \\ \qquad < \eta'' , - > = \mathcal{B}\mathit{ind}(\delta , \lambda' , \eta' , -) \\ < - , [\mu^{m} [\delta] \mu' , - , - , -] , - , `\eta , - , \lambda > \\ \quad \text{otherwise} \end{cases}$$

$S_{\mathcal{P}}$ (PUT_CONSTANT C ac)

$$< - , [\mu^{n}\, x\, \mu' , - , - , -] , - , - , - , - >$$

when ac = A_n (other cases not given in this paper)

$$= \; < - , [\mu^{n} [C] \mu' , - , - , -] , - , - , - , - >$$

$S_{\mathcal{P}}$ (PUT_CONS ac)

$$< - , [\mu , - , \tau , \nu] , \theta , `\eta , - , \lambda >$$

when ac = A_n or not specified
(other cases not given in this paper)

$$= \begin{cases} < - , [\mu'^{n} [\lambda'] \mu'' , \theta , \lambda' , \nu], - , `\eta , - , \lambda' > \\ \quad \text{when ac} = A_n \\ \qquad \text{where } \mu = \mu'^{n}\, x\, \mu'' \\ \qquad \text{and } \lambda' = \mathit{mkcons}(\text{Nil} , \text{Nil} , \lambda) \\ < - , [\mu , - , \lambda'' , \nu] , \theta' , \tau' \rightarrow \eta' , - , \lambda'' > \\ \quad \text{when ac is not specified} \\ \quad \text{where } `\eta = \tau' \rightarrow \eta'' , \\ \qquad \lambda' = \mathit{mkcons}(\text{Nil} , \text{Nil} , \lambda) \\ \qquad < \eta' , \lambda'' > = \mathit{bind}(\tau , \lambda' , \eta'' , \lambda') \\ \qquad \theta' = \mathit{trail}(\tau , \theta , \eta' , \nu) \end{cases}$$

(6) Get operation

$S_{\mathcal{P}}$ (GET_VARIABLE s ac)

$$< - , [\mu , \xi\, x\, \xi'^{n} , - , -] , - , - , - , - >$$

when s = X_n and ac = A_m
(other cases not given in this paper)

$$= \; < - , [\mu , \xi\, \nu\, \xi'^{n} , - , -] , - , - , - , - >$$
$$\text{where } \mu = \mu'^{m}\, \nu\, \mu''$$

$S_{\mathcal{P}}$ (GET_VALUE s ac)

$$< \kappa , [\mu , \xi , - , -] , - , `\eta , - , \lambda >$$

when s = X_n and ac = A_m
(other cases not given in this paper)

$$= \begin{cases} < \kappa , [\mu , \xi , - , -] , - , \tau \rightarrow \eta'' , - , \lambda' > \\ \quad \text{if } \mathit{unify}(\nu , x , \eta' , \lambda) = < \text{success} , \eta'' , \lambda' > \\ \qquad \text{where } \mu = \mu'^{m}\, \nu\, \mu'' , \xi = \xi'\, x\, \xi''^{n} \\ \qquad \text{and } `\eta = \tau \rightarrow \eta' \\ S_{\mathcal{P}}(\text{B backtrack}) < \kappa , [ \mu , \xi , - , - ] , - , `\eta , - , \lambda > \\ \quad \text{otherwise} \end{cases}$$

$S_{\mathcal{P}}$ (GET_CONSTANT C ac)

$$< - , [\mu , - , - , \tau] , \theta , `\eta , \omega , \lambda >$$
$$\text{where } \mu = \mu'^{n} [\tau'] \mu'' , `\eta = \nu \rightarrow \eta' , \text{ and}$$
$$<t , \delta> = \mathit{deref}(\tau' , \eta' , \lambda)$$

when ac = A_n (other cases not given in this paper)

$$= \begin{cases} < - , [\mu , - , - , \tau] , \theta , `\eta , \omega , \lambda > \\ \quad \text{if } t = \text{atom and } \delta = C \text{ (note : C is atom)} \\ < - , [\mu , - , - , \tau] , \theta' , \nu \rightarrow \eta'' , \omega , \lambda' > \\ \quad \text{if } \delta \in \mathbf{Var} \ (t = \text{var...}) \\ \quad \text{where } < \eta'' , \lambda' > = \mathit{bind}(\delta , [C] , \eta' , \omega , \lambda) \\ \qquad \theta' = \mathit{trail}(\theta , \delta , \eta' , \tau) \\ S_{\mathcal{P}}(\text{B backtrack}) < - , [\mu , - , - , \tau] , \theta , `\eta , \omega , \lambda > \\ \quad \text{otherwise } (\delta \neq C) \end{cases}$$

$S_{\mathcal{P}}$ (GET_CONS ac l)

$$< \kappa , [\mu , - , - , -] , - , `\eta , \omega , \lambda >$$
$$\text{where } \mu = \mu'^{n} [\tau] \mu'' , `\eta = \nu \rightarrow \eta'$$
$$< t , \delta > = \mathit{deref}(\tau , \eta' , \lambda)$$

when ac = A_n (other cases not given in this paper)

$$= \begin{cases} S_{\mathcal{P}}(\text{B backtrack}) \\ \quad < \kappa , [\mu , - , - , -] , - , `\eta , \omega , \lambda > \\ \qquad \text{if } t = \text{atom} \\ < \kappa , [ \mu , - , \delta , - ] , - , `\eta , \omega , \lambda > \\ \qquad \text{if } t = \text{list} \\ < \kappa' , [\mu , - , \delta , -] , - , `\eta , \omega , \lambda > \\ \qquad \text{otherwise } (\delta \in \mathbf{Var}) \\ \qquad \text{where } \kappa' = \varphi(l) \end{cases}$$

6. Abstract Symbolic Machine

From definitions 3.3 and 5.3, we observe the following correspondence.

$\mathcal{M}_{\mathcal{L}}$	$\mathcal{M}_{\mathcal{P}}$
K ,	K
R ,	[A*,*H , L , B]
M*,	
C*,*F , W*,*V	C*,*F , W*,*V

We see that

(i) two machines share **C*,*F** , **W*,*V** , and that
(ii) usage of registers are different.
In other words, essential resources of stacks and heaps are shared by the two machines, and the registers used in procedure calls are different besides additional registers **L** and **B** in $\mathcal{M}_P$. Thus when **R** , **M*** and [**A*,*H** , **L** , **B**] are merged, two machines can be integrated. We realize **A*** , ***H** and **M*** on common resources, and make the configuration of ASM as

< **K** , [**R** , **L** , **B** , **A***, ***H**] , **C*,*F** , **W*,*V** > .

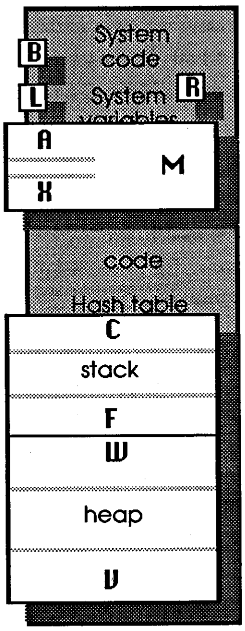

Figure 1. Implemented configuration of ASM

Figure 1 shows how the configuration is realized in real machine (Fujitsu M780). Since **C*,*F** , **W*** and ***V** are shared, there is no logical inconsistency when instruction of $\mathcal{M}_L$ and $\mathcal{M}_P$ are merged. Following instructions of $\mathcal{M}_L$ and $\mathcal{M}_P$ are actually the same or are one of the optional function of the other (inequality means that one includes the other).

$\mathcal{M}_L$		$\mathcal{M}_P$
ENTRY	=	ENTRY
CALL	>	CALL
RTN	=	RTN
DEALLOC	=	DEALLOC
Bcc	>	B
ST	>	CONS_SAVE
L	>	CONS_RESTORE
L	>	TRUST_ME

Moreover, primitives for input/output and storage management (i.e. garbage collection and object allocation) of the heap are shared by the two machines.

The differences in register usages are related to the ways that expressions of Lisp and Prolog are composed. Lisp expressions are recursively composed. This makes it difficult to assign global registers to each arguments in compiling function calls, whereas since Prolog expressions are first-order, it is possible to assign global registers to each argument of goal calls in compiling goal calls.

In ASM, functions and predicates are mutually callable when due considerations are made to assign input/output of values without switching context between Lisp and Prolog which would have been necessary when Prolog and Lisp machines were designed separately.

7. Further work

Because of the limitation of the paper details of the auxiliary functions are not given. Furthermore, we do not discuss how compilers relate meaning of Lisp and Prolog programs to meaning of instructions. Meanings of programs can only be completely specified by giving compilation schemes using the same techniques outlined here. This will be the next theme to pursue.

References

[1] T.Ida, T.Matsuno and A.Nakamura, A practical approach to combining functional and logic programming languages, IFIP Workshop on Concepts and Characteristics of Declarative Systems, Oct., 1988, Budapest
[2] D.H.D.Warren, An abstract Prolog instruction set, SRI International Technical Note 309, 1983.

Data movement	
(L *r* *q* [*m*])	load *q* into *r* [with address modified by *m*]
(ST *r* *w* [*m*])	store *r* into *w* [with address modified by *m*]
Control	
(B*cc* *l*)	branch, *cc* specifies condition codes
(ENTRY)	entry of a procedure
(CALL *id*[*s* [*targ*]])	procedure call to *id* with arguments stored in the frame stack starting from *targ* to *s*
(RTN)	return from a procedure
(DEALLOC *q*)	specify the limit of the current frame by deallocating *q* words of the current frame
(THROW *s*)	execute throw using continuation stored in *s*
(RETURN *s*)	execute return using continuation stored in *s*
(GO *s*)	execute go using continuation stored in *s*
(PUSHCONT*type* *l*)	push continuation of type *type*, *l* is the label at which control is transferred when the continuation is executed by THROW, RETURN or GO instructions.
(UNBINDPOP *n*)	unbind *n* binding pairs stored in the control stack
(CPOP *n*)	pop control stack by *n* words
Arithmetic and logical operations	
(*OP* *r* *q*)	binary operation; *r* and *q* are operands
(*OP* *r*)	unary operation; *r* is an operand
List processing	
(CAR *s* *l*)	take car part of *s*, if *s* is not cons then jump to *l*
(CDR *s* *l*)	similar to CAR
Type check	
(ERROR_CAR *l*)	check whether R is nil or not, If R is nil then jump to *l*, otherwise jump to the appropriate error routine
(ERROR_CDR *l*)	similar to ERROR_CAR
Special variable binding	
(BINDPUSH *s*)	push a binding pair (R , *s*) onto the control stack
Multiple-values	
(SAVE_MV *n* *s*)	save *n* multiple values into the current frame starting at *s*
(RESTORE_MV *n* *s*)	restore *n* multiple values stored in the words starting at *s* in the current frame to the multiple value stack

Note:	*r*	specifies a general register.
	s	specifies a word on the frame stack.
	q	specifies one of *r*, *s* or S-expression.
	w	specifies either *r* or *s*.
	m	specifies addressing mode.
	l	specifies label.
	n	specifies integer

Table 1. Basic instruction set of the Lisp Abstract Machine

note: the instructions given above are slightly different from the 'machine' instructions given in definition 3.1. These instructions are translated to the 'machine' instructions, supplying parameters, if necessary.

Control

(ENTRY)		entry of a procedure
(CALL	*id* [*n*])	procedure call to*id*
(EXECUTE	*id*)	tail recursive procedure call to *id*
(GB	*id*)	jump to the tail recursive entry of the procedure specified by *id*
(RTN)		return from a procedure
(DEALLOC	*q*)	specify the limit of the current frame by deallocating *n* words of the current frame
(CDEALLOC	*q*)	perform DEALLOC only if current frame is top of the frame stack
(B	*l*)*	jump to the location specified by label *l*

Clause group

(TRY_ME_ELSE	*l* [*n*])	set retry continuation for *l*; *n* is the number of active argument registers
(TRY	*l* [*n*])	try *l*, setting retry continuation for the next address
(TRY_ME_ONLY	*l*)	jump to *l*, clearing current retry continuation by GC collectable value
(RETRY_ME_ELSE	*l* [*n*])	reset retry continuation by *l*
(RETRY	*l* [*n*])	reset retry continuation by next address and jump to *l*
(TRUST_ME)		update retry continuation by previous backtrack address (this instruction is used also as cut operator)

Indexing

(SWITCH_ON_TERM	*ac/v lv lc* [*ll*])	access clause groups by type of *ac*
(SWITCH_ON_CONSTANT	*tbl* [*l*])	access to a clause group by the list pointer; jump to *l* if content of the list pointer is not in *tbl*

List pointer movement

(CONS_SAVE	*x*)	save list pointer to *x*
(CONS_RESTORE	*x*)	restore list pointer from *x*

Put operation

(PUT_VARIABLE	*v ac*)	put unbound local variable *v* into *ac*
(PUT_VALUE	*v ac*)	put bound variable *v* into *ac*
(PUT_UNSAFE_VALUE	*y ac*)	put unsafe variable *y* into *ac*
(PUT_CONSTANT	*k ac*)	put constant *k* into *ac*
(PUT_CONS	[*ac*])	allocate a new cons and put it into *ac*

Get operation

(GET_VARIABLE	*v ac*)	set unbound variable *v* to *ac*
(GET_VALUE	*v ac*)	unify bound variable *v* and *ac*
(GET_CONSTANT	*k ac*)	unify constant *k* and *ac*
(GET_CONS	*ac l*)	prepare unification of a cons; jump to *l* if ac is a variable

Note:	*l,lv,lc,ll*	specifies label.
	x	specifies temporary variable.
	y	specifies permanent variable.
	v	specifies *x* or *y*.
	ac	specifies argument register or a car or cdr part of a cons pointed by the list pointer.
	tbl	specifies hash table consisting of labels.
	k	specifies constant.
	n	specifies integer.

Table 2 Basic instruction set of the Prolog Abstract Machine

note: the instructions given above are slightly different from the 'machine' instructions given in definition 5.1. These instructions are translated to the 'machine' instructions, supplying parameters, if necessary.