PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 35, NO. 3, PP. 153-163 (1991)

UPDATE PLANS[1]

H. OSBORNE

Department of Informatics
Faculty of Mathematics and Informatics
University of Nijmegen

Received: Aug. 30, 1990.

Abstract

In recent years many abstract machines have been introduced. In this paper a description language for abstract machines is presented. A formal semantics for this language is defined, and some implementation questions are raised. A few simple examples are given.

The language presented has similarities to graph rewrite systems, and also to classical assembler languages. It could provide a meeting point for formal and implementational specifications.

Keywords: abstract machines, low level implementation, rewrite systems.

1 Introduction

There seems as yet to be no standard notation for describing abstract machines. In functional language circles some form of tuple notation seems to be popular (FAIRBAIRN and WRAY, 1987), an imperative programming style is also often used (LOCK, 1990; JONES and SALKILD) as are informal descriptions (WARREN, 1983). In recent years the Bird-Meertens formalism (BMF, 1989) has been gaining in popularity. Work on abstract machine description in squiggol has been done, and has indeed led to new insights into classes of abstract machines (MEIJER, 1990), but it is still a major step from a squiggol 'programme' to a Von Neumann implementation. Update plans, introduced in (MEIJER, 1986) (on which parts of this article are based), provide a specification method, with sufficient similarity to traditional machine code constructs to be easy to implement, yet of sufficient abstractive power to provide a useful development tool.

The abstractive power is achieved by

- assuming the existence of a separate (static) environment, in which (implementational) details of basic types are specified.
- using only one data structure concept — locations — closely allied to array indexes, addresses and pointers. This, especially, is an im-

[1]Report on the Phoenix project: Esprit Basic Research Action 3147

provement on tuple notations in providing clarity in describing graph like structures. Update plans can indeed also be seen as a conditional graph rewrite system.

- allowing complex state transformations to be described in one transition specification (update scheme).
- factoring out backtracking.

We will not be dealing with this last point in this article. The interested reader is referred to (MEIJER, 1986).

2 An Informal View

Update Plans can be read as describing state transitions in some abstract machine. This machine is considered to consist of some finite number of *stores*, each store containing a countable set of *cells* and addressed by a completely ordered set of *locations*. Conceptually it is not the cells themselves that are addressed, but the boundaries between cells, so that a sequence is not considered to be *at* certain locations, but rather *between* locations. A motivation for this convention is to be found in (MEIJER, 1986). For example, rather than representing the string 'CAT' by an array of characters, and considering the substring 'AT' to be the subarray from location 1 to location 2 one considers 'CAT' to be a sequence of characters between location 0 and location 3, and 'AT' to be the subsequence between locations 1 and 3. The basic notation for such a sequence is

$$0[\text{'CAT'}]3.$$

A singleton sequence is identified with its element. Such a triple

$$\langle location\rangle\,[\,\langle sequence\rangle\,]\,\langle location\rangle$$

is called a *locator expression*. A bag of locator expressions can be used to describe a (sub)configuration of the machine.

The immediate constituent part of an update plan, an *update scheme*, is constructed from two bags of locator expressions forming the *left hand side* and the *right hand side*, and from a boolean expression, known as the *guard*.

$$\langle locator\ \ expression\rangle * :\langle boolean\ \ expression\rangle :\langle locator\ \ expression\rangle *$$

Variables (indicated by lower case words) are allowed in the constituent expressions of an update scheme. An update scheme containing only constants (indicated either by a value or, symbolically, by upper case words)

is known as an *update rule*. Given a substitution mapping variables to (sequences of) constants an update scheme can be instantiated to an update rule.

An update rule is *applicable* to a given configuration if it specifies a subset of that configuration and its guard is TRUE. The result of applying an applicable update rule to a configuration, C, is the superset of the right hand side that is minimally different from C. An update scheme is applicable to a given configuration if there is a substitution that instantiates the scheme to an applicable update rule.

The following example illustrates the concepts introduced above. It is here assumed that the substitution is given, though in practice this is derived from the update scheme and configuration to which it is applied (see subsection 4.5). Consider the configuration (partially) described by the locator expression

$$0[21\ 35\ 27\ 48\ 1]5 \tag{1}$$

and the update scheme

$$k[a]l\ l[b]m : \text{TRUE} : l[b-a]m. \tag{2}$$

Given the substitution

$$\{k \Leftarrow 0, l \Leftarrow 1, m \Leftarrow 2, a \Leftarrow 21, b \Leftarrow 35\} \tag{3}$$

the update scheme in (2) can be instantiated to the update rule

$$0[21]1\ 1[35]2 : \text{TRUE} : 1[14]2$$

which is applicable to configuration (1), and which when applied yields

$$0[21\ 14\ 27\ 48\ 1]5.$$

As mentioned above, an update plan, P, is a bag of update schemes. For any configuration C a *development* of C by P is defined to be

- C, if P contains no update scheme applicable to C
- D, where D is a development of a configuration yielded by application of an instantiation of an element of P to C.

The following plan, taken from (MEIJER, 1986) will develop any configuration having natural numbers x and y between locations A and B and between C and D respectively to one containing the greatest common divisor of x and y between these locations.

$$\begin{array}{llll} A[x]B\ \ C[y]D & : & x > y : & A[x-y]B, \\ A[x]B\ \ C[y]D & : & x < y : & C[y-x]D. \end{array} \tag{4}$$

3 Syntax

3.1 Basic Syntax

The basic syntax for update plans is given by the following context free grammar.

⟨plan ⟩ →
⟨scheme ⟩ *

⟨scheme⟩ →
⟨locator⟩ * : ⟨guard⟩ : ⟨locator⟩ *

⟨guard⟩ →
⟨expression⟩

⟨locator ⟩ →
⟨expression⟩ [⟨expression⟩] ⟨expression⟩ .

⟨expression⟩ →
⟨primary⟩ |
⟨monadic⟩ ⟨expression⟩ |
⟨dyadic⟩ ⟨expression⟩ ⟨expression⟩

⟨primary ⟩ →
⟨constant⟩ |
⟨variable⟩

⟨variable ⟩ →
⟨lower case letter⟩ +

⟨constant ⟩ →
⟨upper case letter ⟩ + |
⟨value⟩

3.2 Syntactic Sugar

Commands

The update schemes in (4) in section 2 are very restricted, in that they can only be applied to numbers appearing between fixed locations. More useful update schemes would also allow one to specify the locations, and might look like

$$\begin{array}{llllllll} A[a]B & C[c]D & a[x]b & c[y]d & : & x > y & : & a[x-y]b, \\ A[a]B & C[c]D & a[x]b & c[y]d & : & x < y & : & c[y-x]d. \end{array} \tag{5}$$

In the context of a larger plan, however, the applicability of such schemes is not easily determinable. The obvious solution is to introduce some form of command stream, and to rewrite (5) to

$$\begin{array}{llllllll} PC[pc]p & pc[GCD & a & c]qc & a[x]b & c[y]d & : x > y & : a[x-y]b, \\ PC[pc]p & pc[GCD & a & c]qc & a[x]b & c[y]d & : x < y & : c[y-x]d. \end{array} \tag{6}$$

Such an update scheme is called a *command*, and may be written as

$$\begin{array}{llllllll} GCD & a & c & a[x]b & c[y]d & : & x > y & : & a[x-y]b, \\ GCD & a & c & a[x]b & c[y]d & : & x < y & : & c[y-x]d. \end{array} \tag{7}$$

More generally, any update scheme exhibiting the pattern

$$PC[pc]p \quad pc[OP \ arg]qc \quad \ldots \quad : \quad \ldots \quad : \quad PC[pc']p \quad pc'[next]qc \quad \ldots .$$

may be rewritten as

$$OP \ arg \quad \ldots \quad : \quad \ldots \quad : \quad next \quad \ldots .$$

Alternative right hand sides

Two update schemes having the same left hand side, such as (7) may be rewritten to share the left hand side, so that (7) becomes

$$\begin{array}{llllllll} GCD & a & c & a[x]b & c[y]d & : & x > y & : & a[x-y]b, \\ & & & & & : & x < y & : & c[y-x]d. \end{array} \tag{8}$$

Omitting locations

It is not always necessary to give both locations in a locator expression. Any location not needed in the dynamic context (see section 4.5) may be

omitted, so that (8) becomes, assuming that the length of a natural number is fixed,

$$\begin{array}{lllll} GCD\ a\ c\ \ a[x]\ \ c[y] & : & x > y & : & a[x-y], \\ & : & x < y & : & c[y-x]. \end{array} \tag{9}$$

Two consecutive sequences may be concatenated. Two expressions $x[s]y$ and $y[t]z$ may then be written as $x[s]y[t]z$ or, if y is not needed in the dynamic context, as $x[s\ \ t]z$.

Omitting guards

A tautological guard may be omitted.

4 Semantics

In the informal presentation in section 2, a store resembled a classical computer memory, consisting of a sequence of cells each containing some object. An equivalent, but for formal specification more useful viewpoint is to see a store as a function from a countable set of locations to a set of objects. Conceptually then, $s\ \ l = o$ is equivalent to 'the cell at location l in store s contains the object o'.

4.1 Configurations

As above, a configuration consists of a finite number of stores. The domains of these stores are pairwise disjoint. All stores have the same range, which is a superset of the union of their domains. For any update plan both the number of stores in a configuration and the range of those stores is fixed.

4.2 Sequences

Since the domain of a store s is countable there is a complete ordering on that domain, which we will assume given, and call $<_s$. The sequence t between locations p and q in store s is defined to be the list of objects $s * [p..q)$, where $[p..q)$ is the list $[p, succ_s\ \ p, \ldots, pred_s\ \ q]$, $succ_s$ and $pred_s$ being the successor and predecessor functions defined by $<_s$. Again, a singleton sequence will, where necessary, be identified with its element. Any resultant ambiguity will be resolved by type information. For example,

a singleton sequence occurring as a location in a locator expression must obviously be interpreted as its element, and not as a sequence.

4.3 Update Rules

The left and right hand sides of an update rule both define partial configurations in a natural way. Consider the locator expression $l[s_0..s_n]r$. This can be interpreted by

$$\mathcal{I}[\![l[s_0..s_n]r]\!] = f \text{ where } f\ \ (l \rightarrow i) = s_i.$$

A bag of constant locator expressions is well formed if the interpretations of its constituent locator expressions are consistent, i.e. if

$$\forall i,j : \mathcal{I}[\![e_i]\!] \supseteq \mathcal{I}[\![e_j]\!]$$

where $\supseteq$ is defined by

$$f \supseteq g \Leftrightarrow \forall l : f\ \ l = \perp \vee g\ \ l = \perp \vee f\ \ l = g\ \ l.$$

As mentioned in section 2, an update rule $(lhs, guard, rhs)$ is applicable to a configuration c if its left hand side is consistent with c (i.e. $\mathcal{I}[\![lhs]\!] \sqsubseteq c$) and its guard is TRUE. The result of updating c by $(lhs, guard, rhs)$ is given by

$$\begin{aligned}\mathcal{A}[\![(lhs, guard, rhs)]\!]\ \ c = c' \\ \text{where } c'\ \ l &= \mathcal{I}[\![rhs]\!]\ \ l, \mathcal{I}[\![rhs]\!]\ \ l \neq \perp \wedge \mathcal{I}[\![lhs]\!] \sqsubseteq c \wedge guard \\ &= c\ \ l, \text{otherwise.}\end{aligned}$$

4.4 Functions

The following functions will be needed in the discussion of the interpretation of update plans. The following conventions apply.

1. $\mathcal{N}$ is the set of natural numbers.
2. $\mathcal{Z}$ is the set of integers.
3. $\mathcal{L}$ is the union of the domains of the stores (locations).

The three functions presented here may be noted both infix and prefix.

$\leftarrow : (\mathcal{N} \times \mathcal{L}) \mapsto \mathcal{L}\ \ n \leftarrow l = pred_s^n\ \ l.$

$\rightarrow : (\mathcal{L} \times \mathcal{N}) \mapsto \mathcal{L}\ \ l \rightarrow n = succ_s^n\ \ l.$

$\leftrightarrow : (\mathcal{L} \times \mathcal{L}) \mapsto \mathcal{Z}\ \ l \leftrightarrow r = n,$ if $l \rightarrow n = r$

$= -n,$ if $n \leftarrow l = r.$

The length function on sequences is noted $\#$. It satisfies $\#\ \ [l..l \rightarrow n) = n$ and $\#\ \ [n \leftarrow l..l) = n$.

4.5 Deriving Dynamic Contexts

Given an update scheme and a configuration, a substitution from variables to (sequences of) constants, such as that given in (3) in section 2 can be derived. This can easily be generalised to provide a method for, given an update scheme, deriving a mapping from configurations to substitutions. In the example above, rather than deriving the substitution (3) in section 2 the dynamic context c defined by

$$\begin{array}{l} c\ config = f \\ \qquad \text{where } f\ l = 1 \\ \qquad\quad f\ m = 2 \\ \qquad\quad f\ a = config\ 0 \\ \qquad\quad f\ b = config\ (f\ l) \end{array}$$

would be derived.

Deriving constraints

Given a locator expression, $e = l[x]r$, and a configuration c, there is a very obvious constraint that any substitution s must satisfy if the instantiation of e by s is to be applicable to c, namely

$$c * \ [s\ l, s\ r) = s\ x.$$

This applies, of course, to all locator expressions appearing on the left hand side of an update scheme, and a set of constraints for any substitution can be found that must be satisfied if the instantiation of the update scheme is to be applicable. This is encapsulated in the following mapping, where again singleton sequences are, where necessary, identified with their elements.

$$\begin{array}{l} \mathcal{C}[\![l1[s1]r1\ \ldots\ ln[sn]rn]\!]\ c \mapsto \quad f \\ \qquad\qquad \text{if } f\ const = const, \text{ if } const \text{ is a constant} \\ \qquad\qquad\quad f\ s1 = c * \ [f\ l1, f\ r1) \\ \qquad\qquad\quad \vdots \quad\ \vdots \qquad\qquad \vdots \\ \qquad\qquad\quad f\ sn = c * \ [f\ ln, f\ rn) \end{array}$$

4.6 Interpretation

Given the obvious semantics of a substitution, the semantics of an update plan can now be defined. For notational convenience we redefine $\mathcal{C}$ to map

from update schemes to substitutions, rather than from bags of locator expressions to substitutions. Here $\mathcal{C}[\![(lhs, guard, rhs)]\!]$ is equivalent to $\mathcal{C}[\![lhs]\!]$ as defined in section 4.5. The set of instantiable update rules for an update scheme s in a configuration c is then given by $(\mathcal{C}[\![s]\!]\ c)\ *s$ and will be, for the sake of brevity, written $s_{\mathcal{C}[\![s]\!]\ c}$. The update of a configuration by a scheme is then simply

$$\mathcal{U}_s\ \ conf\ \ sch = (\mathcal{A}*\ sch_{\mathcal{C}[\![sch]\!]\ conf})\ \ conf$$

and therefore the update of a configuration by a plan is

$$\mathcal{U}_p\ \ plan\ \ conf = \cup/\ \ ((\mathcal{U}_s conf)*\ \ plan)$$

which leads to the following definition of the semantics of an update plan

$$\mathcal{U}\ \ plan\ \ confs = \cup/\ \ ((\mathcal{U}_p\ \ plan)*\ \ confs).$$

The meaning of an update plan p applied to an initial configuration i is then

$$\mathrm{Y}\ \ (\mathcal{U}\ \ p\ \ \{i\}).$$

5 Implementational Aspects

Resolving constraints

Obviously, the derivation method given in section 4.5 can only be a first approximation in any implementation, since the si, li and ri's can all be complicated expressions, containing both constants and variables. Some method is needed for resolving these constraints to some simple substitution function. Since this is simply a question of manipulating symbols, $\mathcal{C}$ is redefined to produce text rather than a function. The interpretation of the text is the obvious one.

$$\begin{aligned}
\mathcal{C}[\![\mathrm{l}_1[\mathrm{s}_1]\mathrm{r}_1\ \ldots\ \mathrm{l}_n[\mathrm{s}_n]\mathrm{r}_n]\!] = \{&(\mathrm{s}_1\ \ ,\ \ \mathrm{c}*\ [\mathrm{l}_1, \mathrm{r}_1))\\
&(\mathrm{l}_1\ \ ,\ \ \leftarrow\ \#\ \mathrm{s}_1\ \ \mathrm{r}_1)\\
&(\mathrm{r}_1\ \ ,\ \ \rightarrow\ \mathrm{l}_1\ \#\ \mathrm{s}_1)\\
&\vdots\ \ \ \vdots\ \ \ \ \ \vdots\\
&(\mathrm{s}_n\ \ ,\ \ \mathrm{c}*\ [\mathrm{l}_n, \mathrm{r}_n))\\
&(\mathrm{l}_n\ \ ,\ \ \leftarrow\ \#\ \mathrm{s}_n\ \ \mathrm{r}_n)\\
&(\mathrm{r}_n\ \ ,\ \ \rightarrow\ \mathrm{l}_n\ \#\ \mathrm{s}_n)\}
\end{aligned}$$

Here the first reduction step has already been executed, deriving expressions for $\mathbf{l}_1 \ldots \mathbf{l}_n$ and $\mathbf{r}_1 \ldots \mathbf{r}_n$. However, $\mathbf{s}_i$, $\mathbf{l}_i$ and $\mathbf{r}_i$ can themselves be complicated expressions, which need to be transformed. The aim is to arrive at a set of constraints of the form $\{(v, expr) | c \in variable\}$. This is achieved by a simple rewriting function, the result of which is then filtered to leave only those constraints of the desired form. The rewriting function $\mathcal{R}$ is defined by

$$\begin{aligned}\mathcal{R}\ c = c \cup\ & \\ & \{(\mathbf{r}, \mathbf{l}) \mid (\mathbf{l}, \mathbf{r}) \in c\} \cup \\ & \{(\mathbf{a}, \overline{\odot}\ \mathbf{r}) \mid (\odot\ \mathbf{a}, \mathbf{r}) \in c\} \cup \\ & \{(\mathbf{a1}, {}^{\circ}\!\oplus\ \mathbf{a2}\ \mathbf{r}), (\mathbf{a2}, \oplus^{\circ}\ \mathbf{a1}\ \mathbf{r}), \mid (\oplus\ \mathbf{a1}\ \mathbf{a2}, \mathbf{r} \in c\}.\end{aligned}$$

where $\odot$ is a monadic function and $\overline{\odot}$ its inverse, and $\oplus$ is a dyadic function and ${}^{\circ}\!\oplus$ and $\oplus^{\circ}$ its left and right inverses, respectively. In the current prototype only those functions having deterministic inverses may be inverted. The resolution of the constraints is the least fixed point of $\mathcal{R}\ c$ filtered to leave only constraints of the desired form.

6 Examples

An update plan for deriving greatest common divisors has already been given. For an example of a non-trivial application of update plans see (MEIJER, 1986). Update plans have also been written for TiM (FAIRBAIRN and WRAY, 1987) and are being developed for other abstract machines. The following examples are intended to illustrate some of the possibilities of update plans.

Semaphores

(Assuming a static context in which n is a natural number)

```
P[s]   s[n+1]   ::   s[n].
V[s]   s[n]     ::   s[n+1].
```

Turing Machines

A classical definition of a Turing machine, such as given in (LEWIS and PAPADIMITRIOU, 1981) usually is of the form

$$(K, \Sigma, \delta, s).$$

where K is a set of states, Σ is the alphabet, s is the initial state, and δ is the transition function, $\delta \subseteq (K \times \Sigma) \rightarrow (K \times (\Sigma \cup \{L, R\}))$. The same machine can be specified by an update plan.

$$
\begin{array}{llll}
K[k] \quad I[p] \quad p[\sigma] & : \delta(k,\sigma) = (l,\varsigma) & : & K[l] \quad\quad\quad\quad p[\varsigma] \\
& : \delta(k,\sigma) = (l,L) & : & K[l] \quad I[1 \leftarrow p] \\
& : \delta(k,\sigma) = (l,R) & : & K[l] \quad I[p \rightarrow 1].
\end{array}
$$

A stack arithmetic machine

$$
\begin{array}{llllll}
PUSH\ x & SP[t] & & :: & SP[s] & s[x]t. \\
POP & SP[s] & s[x]t & :: \ x & SP[t] & . \\
PLUS & SP[s] & s[x]t[y] & :: & SP[t] & t[x+y]. \\
MONMIN & SP[s] & s[x] & :: & & s[-x]. \\
 & \vdots & & & \vdots &
\end{array}
$$

References

BMF (1989): International Summer School on Constructive Algorithmics, 1989 held on Ameland, the Netherlands.

FAIRBAIRN, J. – WRAY, S. (1987): Tim: A Simple, Lazy Abstract Machine to Execute Supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science, 274*. Springer Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo.

LOCK, H. C. R. (1990): An Abstract Machine for the Implementation of Functional Logic Programming Languages. Technical Report, GMD Forschungstelle an der Universität Karlsruhe, Vincenz-Prießnitz-Straße 1, Karlsruhe, BRD. Report on ESPRIT Basic Research Action No. 3147 (the Phoenix Project).

LEWIS, H. R. – PAPADIMITRIOU, C. H. (1981): *Elements of the Theory of Computation*. Prentice-Hall.

MEIJER, H. (1986): *Programmar: A Translator Generator*. PhD Thesis, Universiteit van Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands.

MEIJER, E. (1990): A Taxonomy of Function Evaluating Machines. Technical report, Universiteit van Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands. Report on ESPRIT Basic Research Action No. 3147 (the Phoenix Project).

PEYTON JONES, S. L. P. –SALKILD, J. (1989): The Spineless Tagless G-machine. University College, London.

WARREN, D. H. D. (1983): An abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, Menlo Park, California, USA.

Address:

Hugh OSBORNE
Department of Informatics
Faculty of Mathematics and Informatics
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands