Update Plans: Pointers in Teaching Computer Architecture

Hugh Osborne and Jiří Mencák
School of Computing & Mathematics
University of Huddersfield
Huddersfield HD1 3DH, U.K.
{h.r.osborne,j.mencak}@hud.ac.uk

ABSTRACT

Pointers are intrinsic to Computer Science. Each field of Computer Science seems to use its own more or less ad hoc notation for describing pointers and operations on pointers, thus impeding crossover of students' skills from one area to another.

This paper describes *Update Plans*, a "universal" pointer specification language, and its application to teaching Computer Architectures. Consistent use of Update Plans as a supplement to traditional notations can greatly enhance students' ability to apply skills learned in Computer Architecture courses to other pointer applications, and this is also illustrated.

1. INTRODUCTION

Pointers are innate to computer science in general, and to Computer Architectures in particular. Students will be confronted with pointers even at an introductory level (although possibly implicitly) in discussions of e.g. [indirect] addressing modes, and again at a more advanced level — e.g. [vectored] interrupts. Students often experience difficulty in recognising the same concepts when they encounter then in other areas of Computer Science: e.g. in data structures and in compiler construction to name just two. Each field of computer science seems to have its own notations and conventions for describing pointer structures — e.g. Register Transfer Language in computer architectures, informal diagrams when describing data structures, pseudo-code with explicit pointers in compiler construction. While this may arguably have the advantage of providing notations particularly suited to each application, it does impede a crossover of students' understanding of pointer applications and operations from one subject to another — it is quite common for students to understand the abstract notion of pointers in data structures but to have difficulty in implementing them in a high-level language, let alone relating such an implementation to addresses and indirection in low level code. What is needed is a "universal" pointer notation that can supplement, if not replace, the profusion of conventions currently in use, and which emphasises the "low-level" nature of most pointer operations while still allowing a high level of abstraction in their description.

This paper describes *Update Plans*, a "universal" pointer specification language, its application to teaching Computer Architectures, and its rôle in facilitating crossover of students' skills. The Update Plan formalism can be used to specify a wide range of pointer applications. Abstraction mechanisms within the language allow the appropriate level of information hiding, but in constrast to informal notations the hidden information can be fully recovered from the Update Plan specification.

The remainder of this paper is organised as follows. Section 2 contains a brief introduction to Update Plans. This paper is, however, not intended as a tutorial in Update Plans, and the reader is referred to [4, 3, 5] for more information. Sections 3 and 4 concentrate on the mechanics of Update Plan descriptions in describing and teaching concrete and abstract machine architectures. Section 5 illustrates how the same formalism can be used to describe pointers in abstract data types thus encouraging students to identify the relationship between high level and low level operations involving pointers. These three sections are again not intended as tutorials in the particular pointer operations, but as illustrations of the didactic application of Update Plans. The aim of this paper is to show that Update Plan descriptions are at least, if not more suitable than traditional methods as a tool for teaching pointers. Furthermore they can be used in a wide range of areas while emphasising a Computer Architecture perspective. The overall approach to teaching pointers need not change — a change of tool, rather than a change of method is being proposed here. Section 6 summarises the use of Update Plans as an educational tool in many fields of Computer Science, and discusses possible further developments.

2. UPDATE PLANS

The Update Plans formalism was originally introduced [8] as a 'target language' in the framework of the design of a translator generator. It has since been extended for the general description of machines and algorithms, and as a tool for completion of formal proofs of program implementations [4, 5]. It has also been used as a didactic tool at various universities.

The basic concept underlying the Update Plan formalism is that of an *update* of a machine configuration, each possible update being specified by an *update rule*.

An update plan is a set of *update schemes*, each of which may contain unspecified values. A update scheme containing no unspecified values is called an *update rule*. Update schemes yield update rules by instantiation. A scheme consists of a left-hand side and a right-hand side, both being sets of

locator expressions.

A locator expression is a triple, written $\alpha[\xi]\beta$, where α and β are addresses or *locators* in one of a set of stores in an underlying machine model. Each store is a linear countably infinite sequence of memory cells (e.g. bytes or machine words). The above locator expression expresses the fact that $\alpha \leq \beta$ and that the cells between the addresses α and β contain (a particular representation of) the value of ξ. The notation of a locator expression is chosen such that it looks like the picture $\underset{\alpha}{\lfloor}\quad \xi \quad \underset{\beta}{\rfloor}$.

An update scheme states that if it is applicable (i.e. if all locator expressions in its left-hand side are satisfied), the memory may be minimally updated such that thereafter all locator expressions in its right-hand side are satisfied. The left and right-hand side of an update scheme are separated by an arrow ($\Longrightarrow$) which optionally carries a *guard* ($=\!\![\ \gamma\ ]\!\!\Rightarrow$) which is an additional applicability condition.

For instance, the two-scheme update plan in Figure 1 implements Euclid's algorithm for computing the greatest common divisor of the number initially between `A` and `B` and that initially between `B` and `C`. Capitalised words denote constants: `A`, `B` and `C` are fixed locators and `x` and `y` are unspecified values. In fact, if at any stage of the computation the machine configuration contains `A[9]B` and `B[6]C`, (only) the second update scheme can be instantiated to an applicable update rule, whereupon the `9` is replaced by a `3`. An unspecified value on the left hand side of an update scheme can be considered a variable which obtains its value by means of instantiation.

Simple notational conventions acknowledge the existence of a *programme counter* at a (hidden) fixed locator (by convention `PC`), and allow the omission of irrelevant addresses and the combination of adjacent locator expressions. The update schemes in figure 2 for example, which specify the *push* and *add* instructions on some zero address machine are examples of applications of these conventions. The locator `SP` is the stack pointer.

The expressive power of Update Plans is greatly increased by the use of a macro-like mechanism known as *archetypes.* Using the archetype mechanism complicated pointer structures, families of such structures, and even infinite classes of arbitrarily large structures may be replaced by a single archetype call, thus making it possible to express many update schemes as one.

Archetypes are inspired by macro mechanisms. Their parameter resolution system is purely "macro" in flavour, though their expansion may be context driven, i.e. dependent on the configuration in which the macro is expanded.

An archetype definition defines a left and right hand side in the archetype *body.* When an archetype call is expanded the left and right hand sides of its body are included in the left and right hand sides respectively of the update scheme in which the call appears. There is a parameter resolution mechanism to ensure consistent replacement of archetype parameters.

An example of an archetype definition, and of a possible application is given in Figure 3. The archetype definition (1) defines a `pop` wich pops the value `x` from the stack (addressed through the stack pointer `SP`). In (2) this archetype is applied in an update scheme defining an `ADD` instruction that will pop a value from the stack and add it to the value in the accumulator (`ACC`). Finally (3) shows the same update scheme, but with the `pop` archetype replaced by its expansion. Note that the information hidden by the `pop` archetype has now become explicit.

Larger examples of Update Plan specifications can be found in, e.g., [2, 3, 5].

3. CONCRETE ARCHITECTURES

Consideration of pointers is unavoidable when teaching Computer Architectures. Even the simplest addressing mode — direct addressing — involves a pointer. The *object* represented by a direct addressed operand is the *address* at which the *value* of that operand can be found. For example, in 68K assembler `MOVE 1234, 5678` means "copy the value at address `5678` to address `1234`" — `1234` and `5678` are pointers to the data. More complex addressing modes — e.g. register indirect — can involve multiple indirections and even side effects — e.g. predecrement addressing mode. A popular notation for explaining these indirections and side effects is Register Transfer Language. An example is given in Figure 4 (adapted from [1]). While this notation is reasonably transparent, and is relatively succinct for a single opcode and operand combination, it intermingles the effects of the opcode and the addressing modes, requiring a separate description for each possible opcode and addressing mode, leading to a combinatorial explosion of definitions as the number of instructions and addressing modes increases.

The Update Plan formalism separates the definition of the opcode from the definitions of addressing modes, making it easier to teach these as separate concepts. For example part of the definition of an operand in some assembly language might be as shown in Figure 5. This defines the archetype `oprnd` with two parameters. The first parameter, `ea`, is the *effective address* of the operand; the second, `v` is the value of the operand. Line (4) defines a register indirect addressing mode, with `r` as the register identifier (e.g. `R6`). The locator expression `r[ea]` states that the effective address can be found in this register, while the locator expression `ea[v]` describes the indirection needed to find the value `v` at effective address `ea`. Line (5) provides an alternative definition of an operand, this time in predecrement addressing mode. Again `r` is the register identifier. Access to the effective address and value is similar to the previous case, except that the value in the register must be decreased (moved left across the value `v`) to give the effective address. The locator expression `r[ea]` to the right of the guard ($\Longrightarrow$) expresses the update of the contents of the register. These two lines would be part of a longer archetype definition defining all possible addressing modes in terms of their effective address and value.

Once the addressing modes have been defined they can be used to define the effect of opcodes. For example, the `ADD` opcode is defined in Figure 6. This definition explicitly shows the effect of instruction execution on the programme counter

Figure 1: Euclid's Algorithm for the Greatest Common Divisor, its Implementation in Update Plans, and an Instantiation of an Update Scheme

```
while (x ≠ y) {
  if (x < y) y = y − x;
  else x = x − y;
}
return x;
```

$$\mathtt{A[x]B\ B[y]C} \;=\!\!\{\, \mathtt{x < y} \,\}\!\!\Rightarrow\; \mathtt{B[y-x]C.}$$
$$\mathtt{A[x]B\ B[y]C} \;=\!\!\{\, \mathtt{x > y} \,\}\!\!\Rightarrow\; \mathtt{A[x-y]B.}$$

$$\mathtt{A[9]B\ B[6]C} \Longrightarrow \mathtt{A[3]B.}$$

Figure 2: PUSH and ADD Instructions Specified in Update Plans

$$\mathtt{PUSH\ x\ SP[q]} \Longrightarrow \mathtt{SP[p]\ p[x]q.}$$
$$\mathtt{ADD\ SP[q]\ [x\ y]q} \Longrightarrow \mathtt{SP[p]\ p[x+y]q.}$$

Figure 3: Definition, Application and Expansion of a pop Archetype

(Definition)	$\mathtt{pop(x) = SP[s]\ s[x]t}$	$\Longrightarrow$	$\mathtt{SP[t].}$	(1)
(Application)	$\mathtt{ADD\ pop(x)\ ACC[y]}$	$\Longrightarrow$	$\mathtt{ACC[x+y].}$	(2)
(Expansion)	$\mathtt{ADD\ SP[s]\ s[x]t\ ACC[y]}$	$\Longrightarrow$	$\mathtt{SP[t]\ ACC[x+y].}$	(3)

Figure 4: RTL Definitions for Typical 68K Instructions

```
MOVE  Di,Dj   [Dj]←[Di]
MOVE  P,Di    [Di]←[M(P)]
MOVE  Di,N    [M(N)]←[Di]
EXG   Di,Dj   [Temp]←[Di], [Di]←[Dj], [Dj]←[Temp]
SWAP  Di      [Di(0:15)]←[Di(16:31)], [Di(16:31)]←[Di(0:15)]
LEA   P,Ai    [Ai]←P
```

Figure 5: Definition of an Operand in Update Plans

$$\begin{aligned} \mathtt{oprnd(ea, v)} &= \ldots \\ &= \mathtt{REGIND\ r\ r[ea]\ ea[v]} \Longrightarrow . && (4) \\ &= \mathtt{PREDEC\ r\ r[b]\ ea[v]b} \Longrightarrow \mathtt{r[ea].} && (5) \\ &\vdots \end{aligned}$$

Figure 6: Definition of a Two Address ADD Instruction

$$\mathtt{PC[pc]\ pc[ADD\ op(ea_x, x)\ op(ea_y, y)]qc} \Longrightarrow \mathtt{PC[qc]\ ea_x[x+y].}$$

Figure 7: Alternative Definition of a Two Address ADD Instruction

$$\mathtt{ADD\ op(ea_x, x)\ op(ea_y, y)} \Longrightarrow \mathtt{ea_x[x+y].}$$

(PC). The notational conventions mentioned in Section 2 also allow an alternative form as shown in figure 7 in which the emphasis is on the functionality of the instruction.

By separating the definitions of the opcodes from the definitions of the addressing modes not only has the specification of the instruction set become much more compact and manageable (a simple two address machine with only 16 opcodes and 8 addressing modes which would require 1,024 ($16 \times 8 \times 8$) RTL definitions for a full specification only needs 24 ($16 + 8$) lines in an Update Plan specification) but also it becomes much easier to teach these two concepts independently and in an incremental fashion. Note that the order of presentation of these elements of the instruction set would typically be reversed when presenting them to students. The opcodes can be introduced using a reduced set of addressing modes (e.g. only direct address) as shown in Figure 8, and only when students have mastered this reduced instruction set will the full set of addressing modes be introduced. This approach can be reinforced by the use of a suitable assembler emulator, such as the Postroom Computer [6, 10], which supports this incremental approach. The Postroom Computer is presented to students using the Update Plan formalism to reinforce informal descriptions of the machine. It also uses the Update Plan formalism to describe its internal state when students trace execution of Postroom Computer programmes. Experience has shown that students soon master an (informal) understanding of the meaning of Update Plan notation.

4. ABSTRACT ARCHITECTURES

Pointers are also unavoidable when teaching abstract machine architectures, whether the machine is for a procedural, functional, logical or object oriented language. This section presents an example from an implementation of a functional language. A functional language implementation has been chosen because the data structures in implementations of functional languages are typically small, and very limited in

Figure 8: Definition of a Simple Two Address ADD Instruction

$\texttt{ADD ea}_x\ \texttt{ea}_y\ \texttt{ea}_x[\texttt{x}]\ \texttt{ea}_y[\texttt{y}] \implies \texttt{ea}_x[\texttt{x}+\texttt{y}].$

number. The methods illustrated here could also be applied to more complex abstract machines.

Peyton Jones [7] describes function evaluation by graph reduction using pointer reversal by a combination of informal diagrams and a rather clumsy notation for describing the pointer reversal itself, which uses the implicit function 'Left' which hides an essential indirection, requires inspection of the corresponding diagram to show that the operation can only be applied if the forward pointer 'F' points to an *application node*, and needs an explicit statement of the simultaneity of the components of the pointer reversal operation (see Figure 9 — adapted from [7]). Not only can pointer reversal be expressed much more succinctly in Update Plans (see Figure 10), but all of the hidden information in Figure 9 is now explicitly present in the description.

Note also that the implementational structure of an application node is much clearer in the Update Plan specification — an application node being a structure containing a constant (`APP`) identifying it as an application node, and two pointers (`c` and `d` on the left hand side, `b` and `d` on the right hand side).

Relatively simple Update Plan specifications of the other operations involved in graph reduction can be given, supplementing an informal diagrammatical explanation, and allowing students to experiment with the implementation. A prototype implementation of (a subset of) Update Plans is currently being used in an advanced level course on the implementation of functional languages. Students can, for example, be given Update Plan implementations of standard graph reduction operations and be asked to develop a λ-calculus to graph expression compiler, or they can be asked to develop a compiler from λ expressions to Update Plan specifications of supercombinators.

It should again be emphasised that Update Plan specifications are not proposed as a *replacement* for informal notations, but as a *supplement*. The advantages are threefold. Update Plans have a formal semantics, making specifications precise. The existence of an implementation (albeit currently limited) makes it possible for students to experiment with the construction of graph manipulation primitives. In addition, the students following this course have encountered Update Plans earlier in their studies in an introductory course in Computer Architectures, making the crossover of skills easier.

5. ABSTRACT DATA TYPES

Data structures are another area of Computer Science where pointers are rife — both explicitly in e.g. lists, stacks, queues, trees etc., and implicitly in arrays, records, structures, objects, etc. This is illustrated here by using Update Plans to describe binary trees and operations on them. This example shows how the *single rotate left* operation in AVL trees can be defined using (only) Update Plans. The most common way of explaining such structures and operations is by a combination of informal diagram and (pseudo) code, as shown in Figure 11 (adapted from [9]). In the *single rotate left* operation an unbalanced node having no children on the left, but both a child and a grandchild on the right is balanced by promoting the right hand child to the root node, with the original root node as the new root node's left hand child.

In Update Plans the data structure can be defined as an archetype. Figure 12 contains an example defining a binary tree. The first archetype (6) defines the *abstract* structure of a binary tree. This archetype can be read as: "A binary tree is either the empty tree, or a node containing a key and two subtrees". Archetypes (7) and (8) then define the *concrete* structure of binary trees, defining the empty tree to be the `NULL` pointer, and a node to be a pointer to a data structure containing a key, and two pointers to the node's subtrees. A definition of the single rotate left operation is given in Figure 13. Note that this definition can be read as the textual representation of the tree diagram shown in Figure 11. In other words, this definition of the operation *implicitly* contains the pointers. In contrast to the usual style of explanation of the operation as shown in Figure 11 there is, however, no need to change the representation to make the pointers explicit — simple expansion of the `node` archetypes suffices, as shown in Figure 14. It should be emphasised that this is the *same* update scheme as in Figure 13, only after archetype expansion. The abstract data structure in Figure 13 has been transformed into a concrete representation using pointers while staying in the same paradigm.

6. CONCLUSIONS

The previous three sections have demonstrated the application of Update Plans to teaching Computer Architecture. By using a unified notation that is not only applicable to teaching Computer Architectures, but that is also suitable for describing pointer applications in other areas of Computer Science crossover of students' skills and understanding throughout the curriculum is greatly facilitated. Also, students often find it difficult to relate the high level concept of *pointers* to the low level concept of *addresses*. By using the same formalism to describe both the relationship is made explicit, strengthening students' understanding of pointers from a Computer Systems Architecture perspective.

Update Plans cannot completely replace the other methods discussed here, especially informal graph diagrams — a picture is, after all, worth a thousand words, or even Update Plans — but the formalism should be used to complement and unify explanations of the rôles of pointers in Computer Science and to emphasise the low level nature of most pointer operations.

The Update Plan formalism has been used successfully as a descriptive tool in teaching introductory Computer Architectures. An implementation of a subset of Update Plans is also available, and this is being used to provide more advanced students with hands-on experience — designing and implementing their own instruction sets, and building and using an abstract intermediate code machine.

Figure 9: Pointer Reversal in an Implementation of a Functional Language

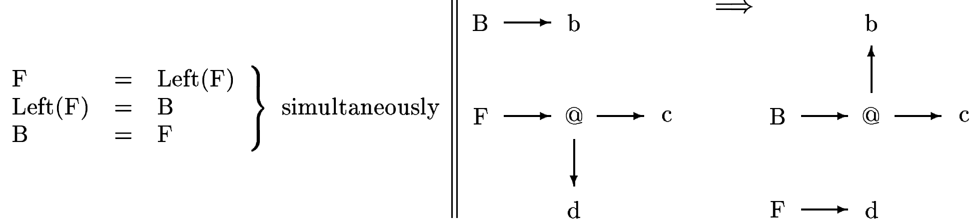

Figure 10: Updata Plan Specification of Pointer Reversal

`F[f] B[b] f[APP c d]` $\Longrightarrow$ `F[c] B[f] f[APP b d].`

Figure 11: Single Rotate Left in an AVL Tree

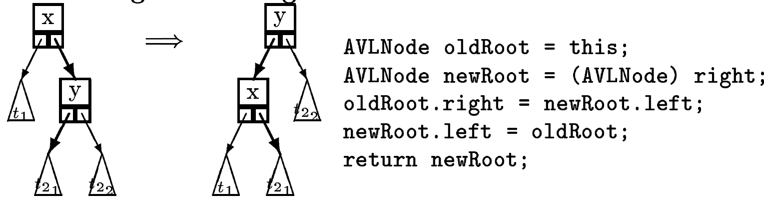

Figure 12: Defining a Binary Tree in Update Plans

$$\begin{aligned} \texttt{tree()} &= \texttt{empty()}. \\ &= \texttt{node(key, tree(), tree())}. && (6) \\ \texttt{empty()} &= \texttt{NULL}. && (7) \\ \texttt{node(key, left, right)} &= \texttt{a a[key left right]}. && (8) \end{aligned}$$

Figure 13: The Single Rotate Left Operation on AVL trees

$$\begin{aligned} &\texttt{ROL oldroot(x, tree()}_1\texttt{, newroot(y, tree()}_{2_1}\texttt{, tree()}_{2_2}\texttt{))} \\ &\Longrightarrow \texttt{newroot(y, oldroot(x, tree()}_1\texttt{, tree()}_{2_1}\texttt{))}. \end{aligned}$$

Figure 14: The Update Scheme from Figure 13 after Expansion of the `node` Archetypes

$$\begin{aligned} &\texttt{ROL oldroot oldroot[x tree()}_1\texttt{ newroot] newroot[y tree()}_{2_1}\texttt{ tree()}_{2_2}\texttt{]} \\ &\Longrightarrow \texttt{newroot newroot[y oldroot tree()}_{2_2}\texttt{] oldroot[x tree()}_1\texttt{, tree()}_{2_1}\texttt{]}. \end{aligned}$$

7. REFERENCES

[1] Alan Clements. *The Principles of Computer Hardware*. Oxford University Press, 2000.

[2] Hugh Osborne. The semantics and syntax of update schemes. In *Code Generation — Concepts, Tools, Techniques*, Workshops in Computing. Springer Verlag, 1992.

[3] Hugh Osborne. Update Plans. In *Proceedings of the 25th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 1992.

[4] Hugh Osborne. *Update Plans — A High Level Low Level Specification Language*. PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1994.

[5] Hugh Osborne. Update Plans for parallel architectures. In M. Kara, J.R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 79–90, Amsterdam, 1996. IOS Press.

[6] Hugh Osborne. The Postroom Computer: Teaching introductory undergraduate computer architecture. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002)*, 2002.

[7] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[8] Hans Meijer. *Programmar: A Translator Generator*. PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1986.

[9] Russel Winder and Graham Roberts. *Developing Java Software*. John Wiley & Sons, 1998.

[10] William Yurcik and Hugh Osborne. A crowd of Little Man Computers: Visual computer simulator teaching tools. In *Proceedings of 2001 Winter Simulation Conference*, New York, 2001. ACM.