J. Functional Programming **6** (2): 329–354, March 1996 © 1996 Cambridge University Press

Formal basis for the refinement of rule based transition systems

A. N. CLARK

Department of Computing, University of Bradford, Bradford, West Yorkshire BD7 1DP, UK
(e-mail: a.n.clark@comp.brad.ac.uk*)*

Abstract

This paper makes a contribution to the refinement of systems which involve search by proposing a simple non-deterministic model for rule based transition systems and using this to define a meaning for rule based refinement which allows each stage of the software development path to be verified with respect to the previous stage. The proposal allows a system which involves search to be specified in terms of all the possible outcomes. Each stage of refinement will introduce complexity to the rules and therefore develop the search space in ever more sophisticated ways. At each stage of the refinement it will be possible to be precise about which collections of outcomes have been deleted, thereby achieving a verified (prototype) implementation.

Capsule Review

One of the neglected application areas for functional programming has been in Artificial Intelligence, despite the obvious applicability of functional programming to the symbolic processing involved, and despite (or maybe because of) the widespread use of LISP in this domain. This paper brings functional programming into an AI application in two ways. Firstly, it brings a formal rule-based analysis to the typical problem of controlling a searching through a space for a solution which matches certain criteria. Traditionally, this is done by heuristics: in this paper a more formal development is presented. Secondly, the implementation of this technique is couched in a monadic style encapsulating the search strategy. Thus this paper is a valuable contribution to bringing functional programming into this exciting application area.

1 Introduction

Many computer programs involve search. A typical scenario is that at various points during the calculation of the program a choice between several alternatives is presented. Some of the alternatives may turn out to be incorrect, since they do not compute an acceptable outcome. For each choice there may be more than one alternative which will compute such an outcome. Often it will only be necessary to select a single alternative at each point in the calculation, but the suitability of an outcome may not depend upon the local context of its calculation.

When implementing programs which involve search, it would be convenient to be able to describe a process which produces all possible outcomes and then applies a filter which throws away those outcomes which do not meet the required acceptance criteria. Unfortunately, a characteristic feature of many programs which involve search is that the resources which would be required to produce all the possible outcomes are at best beyond the scope of most computer systems.

Current practice deals with these problems using 'heuristic' rules which cut down on the amount of outcomes which are produced by selecting one of the alternatives at various choice points in the program. This is done by detecting patterns in the program data which indicate that some of the alternatives will be superfluous in generating acceptable outcomes. These systems have been termed 'Knowledge Based' or 'Expert' (Beynon-Davies, 1993) because of the way they take advantage of information, often attributed to a person who has worked 'in the field' with such systems for years, which indicates the most optimal way of navigating through a graph of choices.

The literature (e.g. Rushby, 1988; Lopez *et al.*, 1990; O'Keefe *et al.*, 1987; Culbert *et al.*, 1987) describes the development of these systems as often being *ad hoc*, where the rules are developed until the system produces an acceptable outcome. In many cases these systems are not specified, but are developed interactively until they produce acceptable outcomes for as many executions as possible. There is no clear development path leading from a specification to an implementation and as a consequence, there is often no way of knowing how complete the implementation is with respect to all the possible acceptable outcomes.

The aims of this work are to propose methods which can be used to increase the overall *quality* of Knowledge Based Systems. To this end, the following objectives have been defined. To propose a simple model for KBS software development which can be used to analyse the features affecting the quality of the system. To propose a simple formalism for expressing Knowledge Based Systems and which is amenable to analysis. To propose characteristic features of the development process which can be controlled effectively. To propose an implementation mechanism for the KBS formalism and to use this as the basis of an example development to give evidence of the utility of the approach.

2 Guide to the paper

This paper is organised as follows: section 3 describes rule based transition systems which are used to express systems involving search. The systems are given a simple semantics by mapping them to a collection of indexed sets of chains and development (or *refinement*) is defined as a rule set transformation which preserves certain properties of its semantics. Section 4 describes how the rule sets are implemented using a monad to deal with the non-determinism which occurs when rule patterns match data values. Section 5 is an example refinement using the Blocks World as a simple system involving search. Finally, section 6 concludes by describing the performance of the mechanisms, related work and further research.

3 Rule based transition systems

Knowledge Based Systems (KBS) (e.g. Lucas and Van Der Gaag, 1991) is a branch of Artificial Intelligence which is concerned with constructing computer programs which perform some task which is traditionally thought of as requiring some degree of human intelligence. For example, medical diagnosis, computer system configuration, action planning and game playing are typical KBS application domains. A number of programming paradigms have been developed which support the construction of KBS software, one of the most widespread of which is *production system* technology (e.g. Luger and Stubblefield, 1989). Typically, a production system involves a collection of data items, referred to as *working memory*, a collection of *rules*, a *control strategy* and a *conflict resolution strategy*. The control strategy defines a mechanism by which a number of rules are selected in order to modify or deduce information from the items in the working memory. Where more than one rule is selected, the conflict resolution strategy is used to reduce the choice to the required amount (usually a single rule). Production systems tend to be characterised by the direction in which the rules are used, two distinct types are *forward-*, e.g. OPS5 (Brownston *et al.*, 1985) and *backward-*, e.g. Prolog (Clocksin and Mellish, 1984), chaining production systems, but in principle there is no limit to the complexity of the control strategy (which can even be implemented in terms of a KBS).

An example of a very simple KBS is the Blocks World (Nilsson, 1980), which consists of a flat surface supporting towers of differently coloured blocks. The aim of a Blocks World system is to move blocks, one at a time, from the top of one tower to the top of another and arrive at an arrangement of blocks in which one of the blocks (the red, one for example) is directly on top of another (the green one, for example). Our claim is that a simple view of KBS can be used as a model for KBS program development. An initial executable specification for the system can often be expressed easily and succinctly using a non-deterministic transition system. In the case of the Blocks World:

(A) Given the current state of the table top, select a block, at random, from the top of a tower and move it to another tower, again selected at random.

When this specification is executed, it will be repeatedly applied to the current system state until the red block is on the green block. Although the specification captures all the required behaviour, it is likely to be very inefficient in practice, since there is no guarantee that a sensible move is made at any given instant, nor is there a guarantee that the execution will terminate. A human would use some 'intelligence' about the moves which are unlikely to succeed and those which step closer to the overall aim. For example:

(B) Given the current state of the table top, if the red and green blocks are free then move the red onto the green otherwise select a block and move it providing that the number of blocks covering red and green are not increased in the process.

We wish to view the development of (B) from (A) as adding 'knowledge' to the initial specification and aim to provide a formalism which will express such systems and allow atomic development steps to be defined and formally justified.

We view KBS implementations as systems which can be described as sequences of transitions. The permissible transitions are described as a collection of rules each of which has an antecedent and consequent pattern. At any given time the system is described as a single value which is referred to as its state. If the state matches any of the antecedent patterns then it makes a transition to the state described by the corresponding consequent pattern. More than one antecedent pattern may match a state and an antecedent pattern may match a state in more than one way. We describe the behaviour of such a system as non-deterministic when the intention is that there is only one outcome from a sequence of transitions and there is no mechanism for controlling which of the many possible outcomes this will be. Such systems have the full power of Turing machines (Jounanaud and Dershowitz, 1990) and hence will support all computations which can be performed by production systems. The relationship between production systems and the transition systems described in this paper is as follows:

- Working memory is represented as the entire state of the non-deterministic transition system.
- Production system rules may involve repeated variables and unification. The transition system rules involve only matching with singly occurring variables.
- Production system control may be quite elaborate whereas the control for the transition system is very simple and corresponds closely to that of forward chaining production systems.
- Production systems may have complex conflict resolution strategies whereas the transition systems use non-deterministic selection.

Although the transition systems which we define are not as expressive as some production systems, this allows the transition systems to have a formal semantics which is essential for the aims of this work. In principle it will be possible to extend the transition systems with more sophisticated features, such as unification.

3.1 Rule sets

Rule sets are collections of rules which consist of antecedent and consequent patterns and a boolean expression. Section 3.1.1 describes the syntax of patterns and how they are matched against data values. Section 3.1.2 describes the syntax and semantics of rule sets.

3.1.1 Patterns

The construction of patterns and the values which they match follows the usual description of *terms* as given in the literature on rewriting systems, with a slight restriction. Let C and I be two disjoint sets, where C contains *function symbols* and I contains *variables*. Each function symbol $c \in C$ has an arity $arity(c)$ and a symbol of arity 0 is called a *constant*. A *term* is either a variable or consists of a function symbol c applied to n terms $c(t_1, \dots, t_n)$ where $arity(c) = n$. A term which contains no variables is a *ground term*. The set of variables which are present in a term t is

$vars(t)$. The transition systems which we describe in this paper can be thought of as rewriting ground terms which we refer to as *values* $v \in V$. An *antecedent pattern* is a term in which each variable may occur only once. A *consequent pattern* is simply a term.

A *substitution* θ is a finite mapping from variables to values. Given a variable $i \in I$ the value of a substitution θ for i is denoted $\theta(i)$. Such a substitution can be uniquely extended to a homomorphism over terms, $\theta(c(t_1, \ldots, t_n)) = c(\theta(t_1), \ldots, \theta(t_n))$, which substitutes for all variables in the term. A substitution may be written $\{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\}$ when $\theta(i_j) = v_j$. A value v *matches* a pattern p when there is at least one substitution such that $\theta(p) = v$.

The values and constructors may be described by an equational algebra (Goguen, 1976), where the equations give rise to the ambiguities as to how a pattern matches a value. For example the equational algebra describing lists: a list of values is either the empty list $[]$, a singleton list $[v]$ or the concatenation of two lists $l_1 +\!\!+ l_2$. The equations define that $[]$ is the left and right identity of $+\!\!+$ and that $+\!\!+$ is associative:

$$\begin{aligned} [] +\!\!+ l = l = l +\!\!+ [] \\ (l_1 +\!\!+ l_2) +\!\!+ l_3 = l_1 +\!\!+ (l_2 +\!\!+ l_3) \end{aligned}$$

For convenience, the following notational sugar is defined in order to construct lists of arbitrary length:

$$[v_1, v_2, \ldots, v_n] = [v_1] +\!\!+ [v_2] +\!\!+ \ldots +\!\!+ [v_n]$$

The predicate *issingletonlist* is true of any list of the form $[v]$. The head and tail of lists are accessed using the following operators: the operator *hd* is defined by $hd([v] +\!\!+ \_) = v$ and the operator *tl* is defined by $tl([\_] +\!\!+ l) = l$. A value v is an element of a list l, $v \in l$, when $l = l_1 +\!\!+ [v] +\!\!+ l_2$ for some lists l_1 and l_2. Notice that $\in$ and the converse $\notin$ are relations which hold both between values and lists and values and sets. A list is reversed using the operator *rev*, i.e. $rev([v_1, v_2, \ldots, v_{n-1}, v_n]) = [v_n, v_{n-1}, \ldots, v_2, v_1]$. The operator *pre* is applied to a function f and a list l to produce a set, $pre(f)(l)$, which contains values which are the result of applying f to all the $+\!\!+$ prefixes of the list l. For example:

$$pre(\mathbf{I})([1,2,3]) = \{[1,2,3], [1,2], [1], []\}$$

Similarly *suff* applies a function to all the suffixes.

Given a value $[1] +\!\!+ [2]$ the pattern $l_1 +\!\!+ (l_2 +\!\!+ l_3)$ will match using the following substitutions:

$$\begin{aligned} &\{l_1 \mapsto [1,2], l_2 \mapsto [], l_3 \mapsto []\} \\ &\{l_1 \mapsto [1], l_2 \mapsto [2], l_3 \mapsto []\} \\ &\{l_1 \mapsto [1], l_2 \mapsto [], l_3 \mapsto [2]\} \\ &\{l_1 \mapsto [], l_2 \mapsto [1,2], l_3 \mapsto []\} \\ &\{l_1 \mapsto [], l_2 \mapsto [1], l_3 \mapsto [2]\} \\ &\{l_1 \mapsto [], l_2 \mapsto [], l_3 \mapsto [1,2]\} \end{aligned}$$

which arise by manipulating the value/pattern in various ways using the two equivalences. This is a special case of unification modulo the theory $A1$ as described in Baader and Siekmann (1994), where A represents associativity and 1 represents an identity element.

The examples in this paper will use the equational algebras for lists and sets. A set of values is constructed from the empty set $\emptyset$, a singleton set $\{v\}$ and set union $s_1 \cup s_2$. The equations define that $\emptyset$ is the left and right identity of $\cup$ and that $\cup$ is both associative and commutative:

$$\begin{aligned}&\emptyset \cup s = s = s \cup \emptyset\\ &s_1 \cup s_2 = s_2 \cup s_1\\ &s_1 \cup (s_2 \cup s_3) = (s_1 \cup s_2) \cup s_3\end{aligned}$$

Matching set patterns is a special case of unification modulo the theory $AC1$ where A represents associativity, C represents commutativity and 1 is an identity element. As described in Baader and Siekmann (1994), $AC1$-unification is generally considered non-trivial due to the rapid growth in complexity when computing unifiers. The problem is greatly reduced by addressing the simpler problem of $AC1$-matching.

For convenience, the folowing notational sugar is defined to construct sets of arbitrary size:

$$\{v_1, v_2, \ldots, v_n\} = \{v_1\} \cup \{v_2\} \cup \ldots \cup \{v_n\}$$

The infix operator $-$ is used for set difference, the predicate *issingletonset* is true of any set of the form $\{v\}$, the operator *element* is defined by $element(\{v\} \cup \_) = v$, the operator *pow* is applied to a function f and a set s and produces a set, $pow(f)(s)$, which is constructed by applying f to each element of the power set of s and the operator *map* is applied to a function f and a set s and produces a set, $map(f)(s)$, which is the result of applying f to each element of s. Notice that these extend the operators described in Goguen (1976) to be higher order.

3.1.2 Rules

A rule r is a pair of patterns and a boolean expression:

$$p_1 \Rightarrow p_2 \ \textbf{when} \ e \qquad \text{where } vars(p_2) \subseteq vars(p_1) \text{ and } vars(e) \subseteq vars(p_1)$$

The rule describes an atomic transformation for a collection of data values. Each value which matches p_1 will give rise to a substitution for the identifiers in p_1. p_2 describes a new data value, given a substitution for its identifiers which must be a subset of those for p_1. A value v which matches p_1 is transformed to a value defined by p_2 and the substitution providing that the boolean expression e holds for v. The following are examples of rules (the boolean expression is omitted when it is the constant *true*):

$$\begin{aligned}&\_ +\!\!\!+\, l \Rightarrow l\\ &l_1 +\!\!\!+ [10] +\!\!\!+ l_2 \Rightarrow l_2 +\!\!\!+ l_1\\ &\{x\} \cup s \Rightarrow s \ \textbf{when} \ x^2 \notin s\\ &\{s_1\} \cup s_2 \Rightarrow s_1 \ \textbf{when} \ s_1 = s_2\end{aligned}$$

The first rule transforms a list by arbitrarily discarding a prefix; the second rule deletes an occurrence of 10 from a list and changes the order of the prefix and suffix of the deleted element; the third rule deletes a number x from a set when the set does not contain the square of the number; finally, the fourth rule produces a set element s_1 when this contains all elements of the set s_2.

The meaning of a rule is given as the set of all pairs (v_1, v_2) such that v_1 matches p_1 and satisfies e and v_2 is the corresponding value which is constructed by p_2. A rule is mapped to its meaning by the operator R which is defined below:

$$R[\![p_1 \Rightarrow p_2 \textbf{ when } e]\!] = \{(v_1, v_2) \mid v_1, v_2 \in V \wedge \theta(p_1) = v_1 \wedge \theta(p_2) = v_2 \wedge e(v_1)\}$$

Rules are collected into sets whose meaning is given by the operator RS:

$$RS(\emptyset) = \emptyset$$

$$RS(\{r\}) = R(r)$$

$$RS(S_1 \cup S_2) = RS(S_1) \cup RS(S_2)$$

A chain c is a sequence of values which is produced by repeatedly applying a rule set to a value. We are interested in all the possible chains which a given rule set can produce. This is generated as the set of chain sets which are indexed by the initial values in the chains. Given a set of value pairs S and a value v then $C(S, v)$ will be the set of all chains which have the initial value v:

$$C(S, v) = \{[v] +\!\!+ c \mid (v, v') \in S \wedge c \in C(S, v')\} \cup \{[v]\}$$

Finally, the meaning of a rule set R is the indexed set of chain sets given by $M(R)$ (which is essentially the same as $\rightarrow^*$ in conventional rewriting theory (Jounanaud and Dershowitz, 1990)):

$$M(R) = \{C(RS(R), v) \mid v \in V\}$$

Given an initial state, a rule set is to be viewed as producing a single chain of transitions non-deterministically selected from the indexed set of chain sets produced by M. Not all of the chains are likely to be acceptable; for example, some do not terminate with a required state and some may be too long. A constraint will generally be specified along with the rule set which dictates which subset of all possible chains are acceptable; the rule set is viewed as producing a chain which is non-deterministically selected from this subset.

3.2 Refinement

The development process from software specification to implementation is generally referred to as *refinement* (Morgan, 1990). The particular formalisms which are used to represent software descriptions and to transform them will differ from application to application, but in general refinement is defined to preserve or achieve various properties at each step in the development process. Morgan (1990) describes a refinement system which is based upon viewing a software system in terms of preconditions on its inputs and postconditions on its outputs, both of which are described using predicate calculus. A refinement step is a modification to the system description which weakens the preconditions and strengthens the postconditions. Such a view of software is very general and abstracts away from the computational mechanisms which are used to compute the specified relations. This paper can afford to take a more restricted view of software since the computational mechanism is known in advance, i.e. rule transitions. The problem which refinement addresses here is not the search for an implementation mechanism, but the *control* of the given implementation mechanism. A specification is viewed as a description of a non-deterministic system without any regard for the efficiency of the calculations which it performs. Refinement is viewed as the process by which a specification is incrementally modified in order to increase the efficiency of the calculations by reducing the number of steps which are taken and reducing the amount of choice available at each step. The refinement which is described in this paper is performed by a human. It is hoped that this work will form the basis of research into machine assisted refinement.

3.2.1 The relation $\leadsto$

A rule set R_1 may be transformed into a rule set R_2 by adding, deleting or modifying rules. The transformation preserves the meaning of R_1 when both rule sets denote the same collection of chains, i.e. when $M(R_1) = M(R_2)$. The transformation is a *refinement* $R_1 \leadsto R_2$ when the collection of chains denoted by R_2 is consistent with those denoted by R_1 in the following way: either the set of chains denoted by R_2 is a subset of that denoted by R_1 or for each R_2 chain which differs, there exists a corresponding R_1 chain which has been shortened to produce it, where a shortening involves deleting an inner subchain. This is made more precise as follows.

The meaning of a rule set, $M(R)$ is defined as a set of all the chains which can possibly be performed when it is used to translate a ground term. The meaning may also be viewed as a directed graph, (N, E, s, t), as follows:

$$t \in N \textbf{ iff } \_ \mathbin{+\!\!+} [(t, \_)] \mathbin{+\!\!+} \_ \in M(R) \vee \_ \mathbin{+\!\!+} [(\_, t)] \mathbin{+\!\!+} \_ \in M(R)$$

$$s(e) = t_1 \wedge t(e) = t_2 \wedge \_ \mathbin{+\!\!+} [(t_1, t_2)] \mathbin{+\!\!+} \_ \in M(R) \textbf{ implies } e \in E$$

$$e_1 \in E \wedge e_2 \in E \wedge t(e_1) = s(e_2) \textbf{ implies } \exists e \in E \bullet s(e) = s(e_1) \wedge t(e) = t(e_2)$$

where each ground term is defined as a node of the graph, each transition is an edge and E is the smallest set satisfying the conditions. A graph also contains the transitive closure of the edges between nodes, where $e_1; e_2$ represents an edge from $s(e_1)$ to $t(e_2)$ when $t(e_1) = s(e_2)$. Graph *homomorphisms* are used to transform one graph into another. Given two graphs, $G_1 = (N_1, E_1, s_1, t_1)$ and $G_2 = (N_2, E_2, s_2, t_2)$, a graph homomorphism $\phi : G_1 \rightarrow G_2$ is a pair of mappings: $\phi_n : N_1 \rightarrow N_2$ and $\phi_e : E_1 \rightarrow E_2$ which obey the following law:

$$\forall e_1 \in E_1 \bullet \phi_n(s_1(e_1)) = s_2(\phi_e(e_1)) \wedge \phi_n(t_1(e_1)) = t_2(\phi_e(e_1))$$

A refinement is defined to be a syntactic transformation on a rule set which is constrained by the following relationship between the semantics of the rule set before and after the transformation. Let R_1 and R_2 be rule sets, the transformation which is applied to R_1 in order to produce R_2 is a *refinement*, $R_1 \rightsquigarrow R_2$, when there exists a graph homomorphism ϕ such that $\phi : M(R_2) \to M(R_1)$, i.e. some possible values have been 'forgotten' (since they are not required); some transitions have been 'forgotten' (since they do not compute a desired result or they are duplicated elsewhere); the result of the modification is *consistent* (when the result does something it agrees with the original rule set); and the number of edges between nodes is reduced (computations between values are made more efficient).

The following is a simple example of a graph transformation for which there exists a homomorphism. The rule set R_1 gives rise to tree-shaped calculations with the states $1, 2, 3, 4, 5, 6, 7$ and transitions a, b, c, d, e, f. Completeness is defined so that any modification must contain states 4 and 7. The rule set R_2 is a modification of R_1 which gives rise to states $1, 2, 4, 5, 6, 7$ and transitions a, b, d, e, g:

$$M(R_1) = \begin{tikzcd} & & 1 \arrow[dl, "a"'] \arrow[dr, "b"] & & \\ & 2 \arrow[dl, "c"'] \arrow[d, "d"] & & 6 \arrow[d, "e"'] \arrow[dr, "f"] & \\ 3 & 4 & & 5 & 7 \end{tikzcd} \qquad M(R_2) = \begin{tikzcd} & 1 \arrow[dl, "a"'] \arrow[d, "b"] \arrow[dr, "g"] & \\ 2 \arrow[d, "d"'] & 6 \arrow[d, "e"'] & 7 \\ 4 & 5 & \end{tikzcd}$$

the modification is a refinement because the following pair of mappings:

$$\phi_n = \{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 6, 7 \mapsto 7\}$$

$$\phi_e = \{a \mapsto a, b \mapsto b, d \mapsto d, e \mapsto e, g \mapsto b; f\}$$

is a homomorphism $\phi : M(R_2) \to M(R_1)$ and the completeness criterion is satisfied. The effect of the modification is to remove state 3, thereby reducing non-determinacy at state 2. Non-determinacy at state 6 is also reduced, seemingly at the expense of increasing non-determinacy at state 1 although on closer inspection this is not the case since the new edge g expands to a pair $b; f$ in $M(R_1)$. The length of the calculation which reaches state 7 is reduced by introducing the new edge g.

The general condition for a modification m between two rule sets R_1 and R_2 being a refinement $R_1 \rightsquigarrow R_2$ is represented by the following diagram:

$$\begin{tikzcd} R_1 \arrow[r, "M"] \arrow[d, "m"'] & G_1 \\ R_2 \arrow[r, "M"'] & G_2 \arrow[u, "\phi"'] \end{tikzcd}$$

where M is the semantic function for rule sets, G_1 and G_2 are the meanings of R_1 and R_2 represented as graphs and ϕ is a graph homomorphism. The modification must also shown to be *complete* with respect to the application specific completeness criteria.

The definition of rule set refinement given above will allow the modifications to increase the efficiency of the rule set whilst enforcing a degree of consistency between the rule sets before and after modification. We are missing a corresponding definition for the *completeness* of the refinement steps, i.e. a constraint which ensures that the modifications do not produce a rule sets which is consistent and more efficient, but which does not perform the required task. This involves the definition of a number of terminal states for the transition system and some proof that the desired terminal states are present after each modification. The completeness definition is less precise than that for refinement since it is likely to differ from application to application.

The refinement relation $\rightsquigarrow$ holds between sets of rules, however, to reduce the amount of unnecessary notation the relation will be used to hold between single rules and between single rules and sets of rules: $r \rightsquigarrow R = \{r\} \rightsquigarrow R$, $R \rightsquigarrow r = R \rightsquigarrow \{r\}$ and $r_1 \rightsquigarrow r_2 = \{r_1\} \rightsquigarrow \{r_2\}$.

3.2.2 Example refinement transformations

To describe a transformation which is a refinement we will place a partial ordering on patterns and prove that transformations which respect the pattern ordering lead to refinements. The reflexive, transitive relation $\sqsubseteq$ between patterns is defined to be the largest relation satisfying the following rules: for any identifier i and composite pattern $c(p_1, \ldots, p_n)$ the following holds $c(p_1, \ldots, p_n) \sqsubseteq i$; for any collection of patterns $p_1 \sqsubseteq p_1' \ldots p_n \sqsubseteq p_n'$ the following holds for any data constructor c, $c(p_1, \ldots, p_n) \sqsubseteq c(p_1', \ldots, p_n')$. The rule set transformations shown in figure 1 all lead to a refinement. Transformation R_1 splits a rule with no constraint into two rules

$$R_1\; p_1 \Rightarrow p_2 \rightsquigarrow \left\{ \begin{array}{l} p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; e \\ p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; \neg e \end{array} \right.$$

$$R_2\; p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; e_1 \rightsquigarrow p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; e_2 \;\; \text{if} \;\; e_2(v) \;\; \textbf{implies} \;\; e_1(v)$$

$$R_3\; \{r_1, \ldots, r_{j-1}, r_j, r_{j+1}, \ldots, r_n\} \rightsquigarrow \{r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_n\}$$

$$R_4\; p_1 \Rightarrow p_2 \rightsquigarrow p_3 \Rightarrow p_4 \;\; \text{providing that} \;\; p_3 \sqsubseteq p_1 \,\&\, p_4 \sqsubseteq p_2$$

$$R_5\; p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; e_1 \;|\; e_2 \rightsquigarrow \left\{ \begin{array}{l} p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; e_1 \\ p_1 \Rightarrow p_2 \;\; \textbf{when} \;\; e_2 \end{array} \right.$$

Fig. 1. Refinement rules.

which have mutually disjoint constraints with the same antecedent and consequent patterns. Any value which matched the antecedent pattern of the original rule will satisfy the constraint on one or other of the refined rules. Transformation R_2 allows the constraints on rules to be 'tightened'. If a rule is transformed by changing the constraint from the boolean expression e_1 to e_2 then e_1 must be true whenever e_2 is for any value v, i.e. $e_2(v)$ **implies** $e_1(v)$. Transformation R_3 allows any rule to be deleted from a rule set. Note that this is only a refinement when the modification is shown to be complete, i.e. to retain the desired outcomes. Transformation R_4 allows the patterns in a rule to be replaced by corresponding patterns which match fewer values, again providing that completeness is established. Transformation R_5 allows a single rule with a disjunctive condition to be replaced by two rules each of which have different halves of the original condition.

The transformations $R_1 - R_5$ are valid when they are shown to uphold the refinement condition in section 3.2.1. In each case the validity is established using the semantic functions R and RS and showing that the chain calculations which are produced by the result of the modification are more efficient and exhibit less non-determinism than those before the modification. The validity of each of the refinement rules is proved in Appendix A as follows: R_1 is valid by theorem A.3, R_2 is valid by lemma A.1, R_3 is valid by the definition of $\rightsquigarrow$ and M, R_4 is valid by lemma A.2 and R_5 is valid by theorem A.2.

The following is a simple example refinement of a single rule:

$$1 \quad \{[x] +\!\!+ l\} \cup s \Rightarrow \{l\} \cup s$$

$$2 \rightsquigarrow \begin{cases} \{[x] +\!\!+ l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \Rightarrow \{l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \\ \{[x] +\!\!+ l\} \cup s \Rightarrow \{l\} \cup s \ \textbf{when} \ 20 \notin l \end{cases} \qquad \text{by } R_1 \,\&\, R_4$$

$$3 \rightsquigarrow \{[x] +\!\!+ l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \Rightarrow \{l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \qquad \text{by } R_3$$

$$4 \rightsquigarrow \{[x] +\!\!+ l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \Rightarrow \{l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \ \textbf{when} \ x \in l_1 +\!\!+ l_2 \ \text{by } R_2$$

$$5 \rightsquigarrow \{[x] +\!\!+ l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \Rightarrow \{l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \ \textbf{when} \ x \in l_1 \qquad \text{by } R_2$$

The rule 1 describes transformations on a set of lists. Given a set of lists, rule 1 will non-deterministically select one of the lists and produce a new set of lists in which the head of the selected list has been discarded. It may be the case that this rule is too general, for example when we want to select the head of only those lists which contain 20. The first stage of the refinement shows rule 1 split into two rules 2 which have mutually exclusive boolean expressions. The two rule sets can be shown to denote the same chains. Next, the second rule at 2 is discarded because for the purposes of our extra constraint it will never be required. This is a genuine refinement because the set of chains denoted by 3 is a subset of those denoted by 2. Next, we restrict ourselves to selecting the head of just those lists which both contain 20 and repeat the head of the list somewhere in the body, and this leads to the refinement at 4. Finally, 5 shows the result of a further constraint which forces the head of the list to occur *before* 20 in the tail.

At each stage in the refinement it is important to prove completeness. The completeness criteria will differ from application to application depending, for example, on whether all of the correct calculations are to be preserved by each modification or whether any one of them will do. Completeness for the example given above could be defined as: it must be possible for *all* lists whose head is repeated before 20 to be transformed; or, it must be possible for *at least one* list whose head is repeated before 20 to be transformed. Given the second definition of completeness, it may be possible to tighten rule 5 further using knowledge about the structure of lists in the set, e.g. 'if candidate lists exist then there will be at least one candidate whose last element is 20', producing the refinement:

$$6 \leadsto \{[x] +\!\!+ l +\!\!+ [20]\} \cup s \Rightarrow \{l +\!\!+ [20]\} \cup s \ \textbf{when}\ x \in l$$

which is complete given the criteria, but will not transform all candidate lists.

4 Implementation

Section 3 defines the syntax and semantics of rule based transition systems and uses these to define the refinement relation $\leadsto$. The semantics is *denotational* and does not help in the execution of rule sets by a computational process. This section describes an *operational* meaning for rule sets using a *monad* to capture the non-determinism which occurs when values are matched against the antecedent patterns of the rules. Section 4.1 gives a brief introduction to monads, section 4.2 defines the monad *ND* which is used to implement pattern matching, section 4.3 defines a compiler which compiles patterns to monad comprehension expressions and section 4.4 discusses the issues which affect the efficiency of pattern matching. The programming language which is used in this section is illustrative, its syntax reflects that of most modern functional languages such as ML (Harper, 1986) and its semantics is call-by-value (Plotkin, 1975) (reflecting its actual implementation in Common Lisp (Steele, 1990)).

4.1 Monads

The following is a very brief introduction to monads and is taken from Wadler (1990), which describes monadic properties in greater detail and gives a large selection of examples. Many functional languages provide a form of *list comprehension* analogous to set comprehension. For example,

$$[(x, y) \mid x \leftarrow [1,2], y \leftarrow [3,4]] = [(1,3),(1,4),(2,3),(2,4)]$$

In general a comprehension has the form $[\iota \mid q]$ where ι is a term and q is a qualifier. A qualifier is either empty Λ; or a generator $x \leftarrow u$ where x is a variable and u is a list-valued term; or a composition of qualifiers (p, q). Comprehensions are translated into expressions which use the operators *unit*, *map* and *join* as defined by the following rules:

$$\begin{array}{lllll} (1) & [\iota \mid \Lambda] & = & unit(\iota) \\ (2) & [\iota \mid x \leftarrow u] & = & map(\lambda x.\iota)u \\ (3) & [\iota \mid (p, q)] & = & join[[\iota \mid q] \mid p] \end{array}$$

Using a type operator M, the type of the three comprehension functions is as follows:

$$\begin{array}{l} map : (\alpha \rightarrow \beta) \rightarrow (M(\alpha) \rightarrow M(\beta)) \\ unit : \alpha \rightarrow M(\alpha) \\ join : M(M(\alpha)) \rightarrow M(\alpha) \end{array}$$

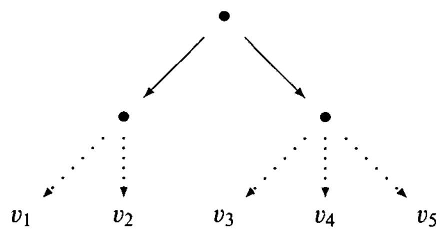
when $M = List$ then we get the list comprehension described above. In general a monad is defined using a type constructor M and a triple of functions map^M, $unit^M$ and $join^M$ which satisfy certain laws which are fully described in Wadler (1990).

4.2 The ND monad

A pattern $l_1 + l_2$ will match a value in a number of different ways. Each different match corresponds to a choice point in the program. Given a list l, the following set comprehension will produce a set of values:

$$[e \mid l_1 \leftarrow pre(\mathbf{I})(l), l_2 \leftarrow suff(\mathbf{I})(l), l_1 + l_2 = l]^{Set}$$

where e is an expression in terms of l_1 and l_2. However, such a comprehension will develop the search space *depth first* which may lead to problems if the wrong alternative is selected. The *ND* monad is used to develop the search space *breadth first*. A typical *ND* value is as follows:



where • represents a choice point in the program, v_1 represents a program value, a full arrow represents a completed evaluation and a dotted arrow represents an evaluation which has yet to occur. Each time an *ND* value is *forced*, all choice points are developed by a single level thereby implementing a breadth first search.

The *ND* monad is implemented in terms of the three operations *unit*, *map* and *join* whose behaviour with respect to typical *ND* values is shown in figure 2. The $unit^{ND}$ operator constructs a singleton *ND* value which immediately produces its result. The map^{ND} operator maps a function f over the leaves of an *ND* value when they are eventually produced. The $join^{ND}$ operator 'flattens' a nested *ND* value by grafting the *ND* values found at the leaves onto the main structure of the parent *ND* value. The *ND* monad is specified by

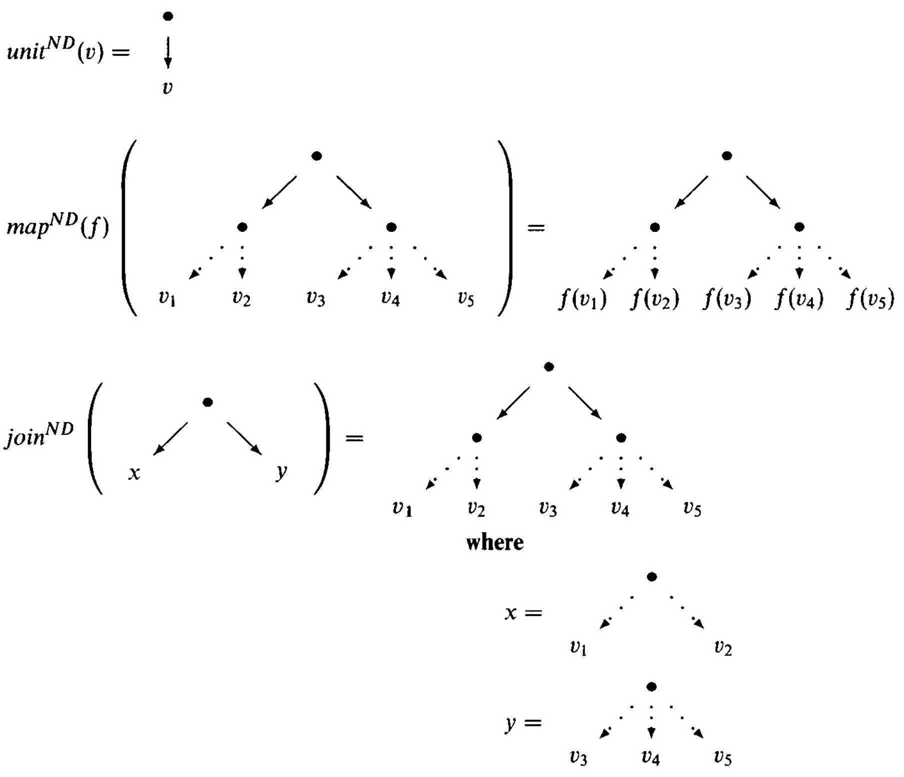
$$\textbf{type } ND(\alpha) = Set(Delay) \textbf{ where data } Delay = just(\alpha) \mid delay(Unit \rightarrow ND(\alpha))$$

$$unit^{ND}(x) = \{just(x)\}$$

$$map^{ND}(f) = leaves(\lambda x.\{delay(\lambda().\{just(f(x))\})\})$$

$$join^{ND} = leaves(\mathbf{I})$$

where the operator *leaves* will lazily map a function f over the leaves of a tree. *leaves* is defined below and uses the operator \ which constructs set homomorphisms



$unit^{ND}(v) =$
v
$map^{ND}(f)$
v_1 v_2 v_3 v_4 v_5
$=$
$f(v_1)$ $f(v_2)$ $f(v_3)$ $f(v_4)$ $f(v_5)$
$join^{ND}$
x y
$=$
v_1 v_2 v_3 v_4 v_5
where
$x =$
v_1 v_2
$y =$
v_3 v_4 v_5

Fig. 2. The *ND* monad operators.

$\backslash(\oplus)(f)(b)$ given a binary operator $\oplus$, a unary operator f and a value b which is the left and right identity of $\oplus$.

```
$leaves(f)(S) =$
    let rec $g(x) =$
        case $x$ of
            $just(v) \Rightarrow f(v)$
            $delay(h) \Rightarrow \{delay(\lambda().\backslash(\cup)g\emptyset(h()))\}$
        end
    in $\backslash(\cup)g\emptyset S$
```

Notice the use of $Unit \rightarrow ND(\alpha)$ which is the type of a delayed expression, or 'thunk', in a strict language such as Common Lisp. This is a trick used in strict functional programming languages to delay the evaluation of an expression, for example if $f(e)$ is a function application expression then ordinarily e would be evaluated, f would be evaluated and then the result of e would be supplied as the argument to the result of f. Changing the expression to $f(\lambda().e)$ causes a function $\lambda().e$ to be supplied as the argument to f; then f can choose whether or not to evaluate e. If f chooses to evaluate e then it will do this by applying the supplied function to the unit value ().

An ND value is a set whose elements are tagged either *just* or *delayed*. A value which is tagged *just* requires no further calculation. An element which is tagged *delayed* is an expression, delayed by turning it into a function of no arguments, which will evaluate to produce an ND value. A search space is developed *breadth first* by delaying each of the choices at every stage. A function *forcefringe* will develop an ND value breadth first until at least one of the values is forced to completion. The function *forcefringe* would be used by a program to continually produce outcomes until the first acceptable one is found.

```
$forcefringe(S) =$
    **if** $any(isjust)(S)$
    **then** $S$
    **else** $forcefringe(forcefringeonce(S))$
        **where** $forcefringeonce(S) =$
            **let** $g(x) =$
                **case** $x$ **of**
                    $just(v) \Rightarrow \{just(v)\}$
                    $delay(h) \Rightarrow h()$
                **end**
            **in** $\backslash(\cup) g \emptyset S$
```

The predicate *isjust* is true of any value of the form $just(v)$, the predicate *any* is applied to a predicate p and a set S and is true when there is at least one element of S which satisfies p.

4.3 Pattern compilation

A rule set is translated to a function which, when applied to an argument, will return a set of outcomes resulting from the multiple matches between the argument and the rule set patterns. The compiler C is used to translate a rule set to the corresponding function; C is also used to translate λ-functions whose formal parameter is a pattern. The semantic brackets $[\![$ and $]\!]$ are used to delimit source code expressions; C is a function which translates source code to source code:

$$C(\{[\![p_1 \Rightarrow p_2 \text{ **when** } e]\!]\}) = [\![\lambda p_1 \textbf{ when } e.p_2]\!]$$
$$C(\emptyset) = [\![\mathbf{K}(\emptyset)]\!]$$
$$C(R_1 \cup R_2) = [\![\lambda x.(f(x)) \cup (g(x))]\!] \textbf{ where } f = C(R_1) \;\; g = C(R_2)$$
$$C[\![\lambda p \textbf{ when } e_1.e_2]\!] = [\![\lambda x.[e_2 \mid q_1, \ldots, q_n, e_1]^{ND}]\!] \textbf{ where } [q_1, \ldots, q_n] = P(p, x)$$

K is the combinator which is defined $\mathbf{K}(x)(y) = x$. A function λp **when** $e_1.e_2$ which contains a pattern p as a formal parameter will be applied as usual to an argument v but will produce more than one result since v may match p in more than one way. The function is translated to have the following form: $\lambda x.[e \mid q_1, q_2, \ldots, q_n]^{ND}$ where the qualifiers $q_1, \ldots, q_n$ are the result of compiling the pattern p using the pattern compiler P, $P(p, x) = [p_1, p_2, \ldots, p_n]$. The compiler P implements both $A1$-

and $AC1$-matching:

$$\begin{aligned}
&\mathit{prefix} = \mathit{pre}(\mathit{just})\\
&\mathit{suffix} = \mathit{suff}(\mathit{just})\\
&\mathit{power} = \mathit{pow}(\mathit{just})\\
&P(i,x) = [[\![i \leftarrow \mathit{unit}^{ND}(x)]\!]]\\
&P(k,x) = [[\![x = k]\!]]\\
&P([\![p_1 +\!\!+ p_2]\!],x) = [[\![\ i_1 \leftarrow \mathit{prefix}(x),\\
&\qquad\qquad\qquad\qquad\quad i_2 \leftarrow \mathit{suffix}(x),\\
&\qquad\qquad\qquad\qquad\quad (i_1 +\!\!+ i_2) = x]\!]] +\!\!+ (P(p_1,i_1)) +\!\!+ (P(p_2,i_2))\\
&P([\![[p]]\!],x) = [[\![\mathit{issingletonlist}(x)]\!]] +\!\!+ (P(p,[\![\mathit{hd}(x)]\!]))\\
&P([\![p_1 \cup p_2]\!],x) = [[\![\ i_1 \leftarrow \mathit{power}(x),\\
&\qquad\qquad\qquad\qquad\quad i_2 \leftarrow \mathit{power}(x),\\
&\qquad\qquad\qquad\qquad\quad (i_1 \cup i_2) = x]\!]] +\!\!+ (P(p_1,i_1)) +\!\!+ (P(p_2,i_2))\\
&P([\![\{p\}]\!],x) = [[\![\mathit{issingletonset}(x)]\!]] +\!\!+ (P(p,[\![\mathit{element}(x)]\!]))
\end{aligned}$$

The pattern compiler P is defined by case analysis on the first argument which is a pattern. If the pattern is i, then the result will bind i to a value non-deterministically selected from x. If the pattern is a constant k then, in order to proceed, the value of x must be k. If the pattern is a concatenation of list patterns $p_1 +\!\!+ p_2$ then two new identifiers are coined, i_1 is bound to a value non-deterministically selected from the prefixes of x, and i_2 is bound to a value non-deterministically selected from the suffixes of x, in order to proceed, the concatenation of the prefix and suffix must be the original list x. This implements $A1$-matching. If the pattern is a singleton list $[p]$ then x must also be a singleton list and p must match the contents of x. If the pattern is a set union $p_1 \cup p_2$ then two new identifiers are coined, both of which are bound to sets non-deterministically selected from the power set of x such that their union is the original set x. This implements $AC1$-matching. Finally, if the pattern is a singleton set, $\{p\}$ then x must also be a singleton set and p must match the contents of x. For example, compilation of the rule:

$$\{[x] +\!\!+ l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \Rightarrow \{l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \ \ \textbf{when}\ \ x \in l_1$$

produces the following function:

$$\begin{aligned}
\lambda i_1.[\{l_1 +\!\!+ [20] +\!\!+ l_2\} \cup s \ \mid\ & i_2 \leftarrow \mathit{power}(i_1), s \leftarrow \mathit{power}(i_1), i_2 \cup s = i_1,\\
& \mathit{issingletonset}(i_2), i_3 \leftarrow \mathit{unit}^{ND}(\mathit{element}(i_2)),\\
& i_4 \leftarrow \mathit{prefix}(i_1), i_5 \leftarrow \mathit{suffix}(i_1), i_4 +\!\!+ i_5 = i_1,\\
& \mathit{issingletonlist}(i_4),\\
& l_1 \leftarrow \mathit{prefix}(i_5), i_6 \leftarrow \mathit{suffix}(i_5), l_1 +\!\!+ i_6 = i_5,\\
& i_7 \leftarrow \mathit{prefix}(i_6), l_2 \leftarrow \mathit{suffix}(i_6), i_7 +\!\!+ l_2 = i_6,\\
& \mathit{issingletonlist}(i_7), 20 = \mathit{hd}(i_7)]^{ND}
\end{aligned}$$

4.4 Efficiency issues

As it stands, the pattern compiler P produces code which is very inefficient. P may be made significantly more efficient by anticipating certain types of patterns which will occur. For example, looking ahead with respect to the pattern $[p_1] +\!\!+ p_2$

reveals that there is no point in generating prefixes which are not singleton lists. Furthermore, there is only one singleton list prefix which can possibly match p_1 and that is the head of the list. The following is an example of how P can be modified to be more efficient:

$$\begin{aligned} P([\![[p_1] +\!\!+ p_2]\!], x) = [[\![& i_1 \leftarrow \{just(hd(x))\}, \\ & i_2 \leftarrow \{just(tl(x))\}]\!]] +\!\!+ (P(p_1, i_1)) +\!\!+ (P(p_2, i_2)) \end{aligned}$$

A similar example occurs when single elements are extracted from sets

$$\begin{aligned} P([\![\{p_1\} \cup p_2]\!], x) = [[\![& i_1 \leftarrow map(just)(x), \\ & i_2 \leftarrow unit^{ND}(x - \{i_1\})]\!]] +\!\!+ (P(p_1, i_1)) +\!\!+ (P(p_2, i_2)) \end{aligned}$$

If the pattern compiler can proceed further and identify that the pattern p_1 is a constant then the following code can be produced

$$P([\![\{k\} \cup p]\!], x) = [[\![k \in x]\!]] +\!\!+ (P(p, [\![x - \{k\}]\!]))$$

In general the further P looks ahead to anticipate unnecessary work, the more efficient the resulting code will be.

5 An example refinement

Section 3.1 describes the semantics of rule sets and defines the refinement relation $\leadsto$ between rule sets. Section 4 describes how the rule sets are implemented using a monad which deals with the non-determinism. This section gives an example rule set refinement, starting with a very simple rule set which serves as a specification and producing a more complex rule set which serves as a prototype implementation.

The example scenario is the Blocks World (Nilsson, 1980) which consists of stacks of blocks arranged on a flat surface. The goal is to place one of the blocks onto another block using simple steps which involve moving a single block from one stack to another. The Blocks World is modelled as a set of lists of integers where the goal is always to place block 1 onto block 2. A single transition initially involves removing the head of one of the lists and 'pushing' it onto the head of another list. The description will always contain a single occurrence of the empty list which represents the flat surface.

For example an initial state is $\{[3,1,4],[5,2,6]\}$ and one possible goal state which can be reached is $\{[1,2,6],[4],[3,5]\}$ where the individual steps were: move 5 onto the flat surface, move 3 onto 5 and finally move 1 onto 2. Obviously most starting states give rise to many different goal states and there are multiple ways of reaching each single goal state from the same starting state.

Figure 3 shows the refinement of the Blocks World rule set. The remainder of this section describes each step in the refinement in detail. Completeness for the Blocks World is that the rule set must always generate chain of transitions from any given starting state to a goal state which is no longer than any other possible chain.

Rule set B_1 shows the starting point for the refinement. The single rule describes the transition which will non-deterministically select two stacks of blocks and move the head of one block to the head of the other. By repeatedly applying this rule

$$B_1 \quad \{[b] \mathbin{+\!\!+} l_1\} \cup \{l_2\} \cup s \Rightarrow \{l_1\} \cup \{[b] \mathbin{+\!\!+} l_2\} \cup s$$

$$B_2 \leadsto \left\{ \begin{array}{l} \{[b] \mathbin{+\!\!+} l_1\} \cup \{l_2\} \cup s \Rightarrow \{l_1\} \cup \{[b] \mathbin{+\!\!+} l_2\} \cup s \\ \quad \textbf{when} \ ((b = 1) \,\&\, (hd(l_2) = 2)) \mid \\ \qquad (((1 \in l_1) \mid (2 \in l_1)) \,\&\, \\ \qquad ((1 \notin l_2) \,\&\, (2 \notin l_2))) \\ \\ \{[b] \mathbin{+\!\!+} l_1\} \cup \{l_2\} \cup s \Rightarrow \{l_1\} \cup \{[b] \mathbin{+\!\!+} l_2\} \cup s \\ \quad \textbf{when not} \left(\begin{array}{l} ((b = 1) \,\&\, (hd(l_2) = 2)) \mid \\ ((1 \in l_1 \mid 2 \in l_1) \ \& \\ \quad ((1 \notin l_2) \,\&\, (2 \notin l_2))) \end{array} \right) \end{array} \right.$$

$$\begin{array}{l} B_3 \leadsto \{[b] \mathbin{+\!\!+} l_1\} \cup \{l_2\} \cup s \Rightarrow \{l_1\} \cup \{[b] \mathbin{+\!\!+} l_2\} \cup s \\ \qquad \textbf{when} \ ((b = 1) \,\&\, (hd(l_2) = 2)) \mid \\ \qquad\quad (((1 \in l_1) \mid (2 \in l_1)) \ \& \\ \qquad\quad ((1 \notin l_2) \,\&\, (2 \notin l_2))) \end{array}$$

$$B_4 \leadsto \left\{ \begin{array}{l} \{[1] \mathbin{+\!\!+} l_1\} \cup \{[2] \mathbin{+\!\!+} l_2\} \cup s \Rightarrow \{l_1\} \cup \{[1,2] \mathbin{+\!\!+} l_2\} \cup s \\ \\ \{[b] \mathbin{+\!\!+} l_1\} \cup \{l_2\} \cup s \Rightarrow \{l_1\} \cup \{[b] \mathbin{+\!\!+} l_2\} \cup s \\ \quad \textbf{when} \ ((1 \in l_1) \mid (2 \in l_1)) \ \& \\ \qquad ((1 \notin l_2) \,\&\, (2 \notin l_2)) \end{array} \right.$$

$$B_5 \leadsto \left\{ \begin{array}{l} \{[1] \mathbin{+\!\!+} l_1\} \cup \{[2] \mathbin{+\!\!+} l_2\} \cup s \Rightarrow \{l_1\} \cup \{[1,2] \mathbin{+\!\!+} l_2\} \cup s \\ \\ \{[b] \mathbin{+\!\!+} l_1 \mathbin{+\!\!+} [1] \mathbin{+\!\!+} l_2\} \cup \{l_3\} \cup s \Rightarrow \{l_1 \mathbin{+\!\!+} [1] \mathbin{+\!\!+} l_2\} \cup \{[b] \mathbin{+\!\!+} l_3\} \cup s \\ \quad \textbf{when} \ 2 \notin l_3 \\ \\ \{[b] \mathbin{+\!\!+} l_1 \mathbin{+\!\!+} [2] \mathbin{+\!\!+} l_2\} \cup \{l_3\} \cup s \Rightarrow \{l_1 \mathbin{+\!\!+} [2] \mathbin{+\!\!+} l_2\} \cup \{[b] \mathbin{+\!\!+} l_3\} \cup s \\ \quad \textbf{when} \ 1 \notin l_3 \end{array} \right.$$

$$B_6 \leadsto \left\{ \begin{array}{l} \{[1] \mathbin{+\!\!+} l_1\} \cup \{[2] \mathbin{+\!\!+} l_2\} \cup s \Rightarrow \{l_1\} \cup \{[1,2] \mathbin{+\!\!+} l_2\} \cup s \\ \\ \{[b] \mathbin{+\!\!+} l_1 \mathbin{+\!\!+} [1] \mathbin{+\!\!+} l_2\} \cup \{l_3\} \cup s \Rightarrow \{[1] \mathbin{+\!\!+} l_2\} \cup \{rev(l_1) \mathbin{+\!\!+} [b] \mathbin{+\!\!+} l_3\} \cup s \\ \quad \textbf{when} \ 2 \notin l_3 \\ \\ \{[b] \mathbin{+\!\!+} l_1 \mathbin{+\!\!+} [2] \mathbin{+\!\!+} l_2\} \cup \{l_3\} \cup s \Rightarrow \{[2] \mathbin{+\!\!+} l_2\} \cup \{rev(l_1) \mathbin{+\!\!+} [b] \mathbin{+\!\!+} l_3\} \cup s \\ \quad \textbf{when} \ 1 \notin l_3 \end{array} \right.$$

Fig. 3. The refinement of the Blocks World rule set.

to a given state, all possible sequences of actions have been described. If this rule is applied blindly (or *stupidly*†) then many undesirable actions will be made. For example, there is nothing to stop a system which employs this rule from getting into a loop by repeatedly swapping a block from the top of one stack to another and then back again.

† Knowledge Based System specifications which are complete with respect to the possible outcomes and use no cleverness to prune the search space, lead to the pun 'Stupidity Based Systems'.

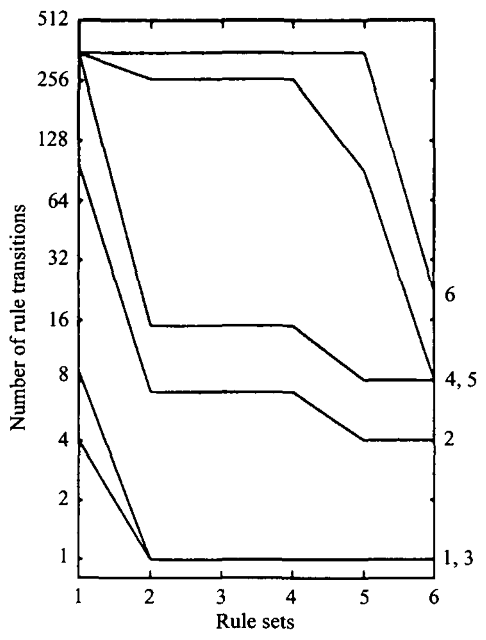


Fig. 4. Increase in efficiency due to refinement.

Rule set B_1 is refined to produce rule set B_2. The first rule deals with the transitions which place block 1 onto block 2, clear block 1 and clear block 2. The second rule deals with all other transitions. Verification follows from the proof of validity for refinement rule R_1.

Rule set B_2 is refined to produce rule set B_3. The refinement step drops the second rule from rule set B_2. Verification must show that the intended goal chains are still present in the denotation of the rule and that the rule has not introduced any new chains. Verification follows from the validity of refinement rule R_3 and theorem A.4.

Rule set B_3 is refined to produce rule set B_4. The refinement step separates out the disjunction from rule set B_3 to produce two distinct rules. Verification follows from the validity of refinement rules R_4 and R_5.

Rule set B_4 is refined to produce rule set B_5. The refinement step separates out the disjunction and absorbs the conditions $1 \in l_1$ and $2 \in l_1$ into the patterns for the respective new rules. Verification follows from the validity of the refinement rules R_4 and R_5.

Finally rule set B_5 is refined to produce rule set B_6. The refinement step absorbs repeated application of the same rules by performing the sequence of transitions in one single jump. Verification follows from the proof of theorem A.5.

Figure 4 shows the result of executing the Blocks World scenario with the different rule sets and starting states of varying complexity. A cut off point was imposed at 350 transitions so the rule sets B_1 – B_5 did not reach a goal state for starting state

6. The starting states are as follows:

(1) $\{[1],[2]\}$
(2) $\{[3,1],[4,2]\}$
(3) $\{[1],[2],[3]\}$
(4) $\{[3,1],[4,2],[5,6]\}$
(5) $\{[3,4,1],[5,6,2],[7,8,9]\}$
(6) $\{[3,4,5,6,7,8,1,9,10],[11,12,13,14,15,2,16,17],$
$[18,19,20],[21,22,23],[24,25,26]\}$

The graph shows that the number of transitions required to find a solution grows dramatically for the unsophisticated rule sets as the complexity of the starting states increases. The starting states 1 and 3 perform very well with all the rule sets because they are both very close to a solution. The identification of the conditions under which a block should be moved which is represented by the refinement $B_1 \rightsquigarrow B_2$ dramatically reduces the number of transitions required for starting states 1,2,3 and 4. On the whole, rule set B_1 performs very badly as expected, but represents a conveniently succinct specification of the Blocks World scenario. Rule set B_6 performs well, is fairly useless as a specification but is shown to be a valid refinement of B_1.

In general, the initial system description is intended to be clear and concise but is allowed to be inefficient. The development process applies formally justified transformations with the aim of increasing efficiency by throwing away unwanted paths in the calculation. The level of inefficiency present in the initial implementation and the required level of efficiency in the final implementation will depend upon the application and the knowledge of the application domain which is used to drive the transformations. The proposed approach to development does not guarantee that an acceptable implementation can be produced, however it does provide a formal basis for analysing the initial (executable) specification with respect to the required reduction in non-determinism and thereby assessing the risk associated with development. For example, the initial Blocks World rule set could be analysed in order to determine how many steps there are to an outcome in the best and worst cases, where a step is defined as the comparison of two atomic data items, the selection of an immediate sub-term from a composite term, the construction of a term from its sub-terms, etc. These steps can be translated into resource usage with respect to a particular programming language. A given refinement rule can be assigned a resource gain determined by the number of steps which it removes from a given rule set. A specification could include the initial efficiency measure and the desired efficiency goal and risk analysis would estimate the feasibility of the refinement. This is an area for future work which would contribute to the area of quality control for KBS development.

6 Conclusion

The aims of this work were defined as proposing methods which contribute to the overall quality of KBS development. This has been achieved by meeting the following objectives. A syntax and semantics for rule sets has been defined which allows programs involving search to be expressed. Development has been formally defined as a refinement relation in terms of syntactic transformations which preserve semantic properties. The semantics of the rule sets is *denotational* since it describes an indexed set of rule chains but does not define how the chains are computed. Monad comprehensions have been used to give an operational semantics to the rule sets which describes how the pattern matching and non-determinism is performed. An example refinement has been presented, the transformations proved correct and shown to increase the efficiency of the solution. This work is novel since it proposes very expressive mechanisms from term rewriting theory as a basis for KBS software and focusses on a formal definition of the KBS development process. The aims have been met since a formal definition of software development allows quality control procedures to be precisely defined and executed.

The performance of the implementation using monads in Common Lisp is fairly slow. This is partly due to the simplicity of the implementation which does not attempt to optimise the code produced by pattern compilation. It is likely that a refinement process would provide a prototype which would be used as a design for a hand coded implementation. The implementation mechanism which is described in §4 uses the trick of hiding an expression behind $\lambda()$. in order to delay its evaluation in a language which eagerly evaluates expressions. Such a trick would not be necessary in a lazy language such as Haskell (Hudak *et al.*, 1992) where evaluation is performed only when necessary in order to compute an output.

This work is closely related to the area of rewrite systems which involves a collection of directed rules which are successively applied to terms in order to transform them into a normal form. A introduction to rewrite systems and a survey of the available literature is found in Jounanaud and Dershowitz (1990). In particular, the system described in this paper is related to conditional ground rewrite systems which involve matching rather than general unification and allow guards on the rewrite rules. The system is also intimately related to rewrite systems which incorportate equations into the rewriting machinery, such as the R/S systems described in Jounanaud and Dershowitz (1990), where the equations define associativity, commutativity and identity. The following are important differences between rewrite systems and the mechanism described in this paper:

- Rewrite systems are primarily concerned with transforming terms into a normal form from which there are no applicable rewrites. In general, this is not true of the rules which are used in Knowledge Based Systems, for example the Blocks World rules need not terminate when the goal state is achieved.
- Rewrite systems are often concerned with proving termination. We have not addressed this issue with respect to proving that goal states are reachable. The issue of specifying Knowledge Based Systems and proving that the implementation in terms of a transition system will reach a goal in a finite number of moves is an area for further work.
- Rewrite systems use the full power of unification to perform transformations. The transition system for the Blocks World requires only simple matching.

Another area for future work is to generalise the mechanisms given in this paper to allow multiple occrrences of the same variable in an antecedent pattern and to investigate the requirements for unification.

Embedding equations into a matching framework is also closely related to E-unification (Baader and Siekmann, 1994), where unification is extended to include equations defining, for example, associativity, commutativity and identity. As described above, the application area addressed by this paper does not require unification, but it is an area for further work to investigate the requirements of more sophisticated applications.

This work is broadly related to the area of program transformation, where Partsch (1983) and Luqi (1993) cover the general issues and Bird (1987) and Burstall and Darlington (1977) are more in the spirit of the transformations described here. We have viewed the specification of rule based transition systems as computational processes from the start, whereas Morgan (1990) gives a different perspective of specification and refinement.

The pattern compilation which is described in this paper is a novel technique; see Peyton Jones (1987) for a more conventional approach to the compilation of patterns which match deterministically.

Nilsson (1980) and Charniak *et al.* (1987) describe various search related applications for computer programs, in particular Charniak *et al.* (1987) describe the connection between search problems and non-deterministic computational processes. This paper describes a simple method of adding non-determinism to a functional program using monad comprehensions, see Sondegaard (1992) for a discussion of the issues involved in non-deterministic functional programs and Clark (1994) for an operational description of a system with builtin non-determinism.

There is very little reported work describing the formal development of systems which involve rules, see Roman *et al.* (1993) and Krause *et al.* (1993) for further discussion. Major *et al.* (1991) describe techniques which make programs involving search more efficient, it would be very interesting to see if refinement as described in this paper could be applied to these techniques.

In general, systems which provide support for the development of KBS software employ complex control strategies and provide expressive rule languages which include packaging rules into modules and meta-rule facilities. The system which is described in this paper is very simple and does not provide such facilities, however the simplicity has allowed the system to be given a formal semantics and even without sophisticated features the rule language is capable of expressing a large class of interesting systems as characterised by the Blocks World example. Using the technology developed in this paper as a basis, the following extensions will provide facilities which are found in conventional KBS development systems: extending the pattern matching to allow repeated variables or unification which will allow much more expressive rule sets; providing a mechanism for formal conflict resolution which controls the non-deterministic behaviour of the rule sets; allowing rules to be grouped into modules and defining a collection of operators to modify and combine rule modules; allowing the overall control strategy of the rule systems to be formally defined (possibly as another (meta-)rule system).

Further work is required to investigate the possibilities of clever pattern compilation for a general class of equational algebras. This paper has described a refinement technique which is entirely performed and checked by a human. It is possible that part of the process can be checked or even performed by a computer program. A number of rule set transformations have been identified, a tool could be developed which suggests likely refinement rules to apply at any given stage and applies them under the direction of the operator. Although the operational description of rule sets has not been proved correct with respect to the denotational description, the implementation which lead to the results in figure 4 gives confidence in its validity.

The example rule set which has been refined is very simple. Further work is required to investigate how generally applicable this technique is to systems which are termed 'Knowledge Based' and 'Expert'.

Acknowledgements

This work was carried out as part of the current collaborative EUCLID RTP 6.3 project MOSES (Military Operational Software for Expert Systems) which is aimed at providing a basis for improved military information systems by the application of knowledge engineering methods, standards and tools. The author wishes to thank the three anonymous referees and Professor Simon Peyton Jones for their helpful comments.

A Proofs of theorems

Define the set of all values which can match patterns as follows:

$$V = \{c(v_1, \ldots, v_n) \mid v_i \in V, c \in C, arity(c) = n, n \geq 0\}$$

where C is a set of data constructors.

Lemma A.1

$$(\forall v \in V \bullet e_2(v) \textbf{ implies } e_1(v)) \textbf{ implies } R(p_1 \Rightarrow p_2 \textbf{ when } e_2) \subseteq R(p_1 \Rightarrow p_2 \textbf{ when } e_1)$$

Proof: Let $(v_1, v_2) \in R(p_1 \Rightarrow p_2 \textbf{ when } e_2)$
then $e_2(v_1)$ due to the definition of R
then $e_1(v_1)$ since e_2 implies e_1
so $(v_1, v_2) \in R(p_1 \Rightarrow p_2 \textbf{ when } e_1)$

Lemma A.2
$p_1 \sqsubseteq p_3 \,\&\, p_2 \sqsubseteq p_4$ **implies** $R(p_1 \Rightarrow p_2) \subseteq R(p_3 \Rightarrow p_4)$

Proof: Since pattern matching is not unification we can disregard the identifiers so long as $vars(p_2) \subseteq vars(p_1)$ and $vars(p_4) \subseteq vars(p_3)$. Define G as follows:

$$G(i) = V$$
$$G(c(p_1, \ldots, p_n)) = \{c(v_1, \ldots, v_n) \mid v_1 \in G(p_1) \wedge \ldots \wedge v_n \in G(p_n)\}$$

Now show by induction that $p_1 \sqsubseteq p_2$ **implies** $G(p_1) \subseteq G(p_2)$

$$\begin{array}{ll}
(1) & c(p_1,\ldots,p_n) \sqsubseteq i \\
& G(c(p_1,\ldots,p_n)) = \{c(v_1,\ldots,v_n) \mid v_i \in G(p_i)\} \\
& G(i) = \{c(v_1,\ldots,v_n) \mid v_i \in V\} \cup V \\
& \text{by definition } G(p) \subseteq V \\
(2) & c(p_1,\ldots,p_n) \sqsubseteq c(q_1,\ldots,q_n) \text{ when } p_i \sqsubseteq q_i \\
& G(c(p_1,\ldots,p_n)) = \{c(v_1,\ldots,v_n) \mid v_i \in G(p_i)\} \\
& G(c(q_1,\ldots,q_n)) = \{c(v_1,\ldots,v_n) \mid v_i \in G(q_i)\} \\
& \text{by induction } p_i \sqsubseteq q_i \textbf{ implies } G(p_1) \subseteq G(q_i)
\end{array}$$

Theorem A.1

$$\left(\begin{array}{c}(\forall v \in V \bullet e_2(v) \textbf{ implies } e_1(v))\,\& \\ p_1 \sqsubseteq p_3\,\& \\ p_2 \sqsubseteq p_4\end{array}\right) \textbf{ implies } \left(\begin{array}{c}R(p_1 \Rightarrow p_2 \textbf{ when } e_2) \\ \subseteq R(p_3 \Rightarrow p_4 \textbf{ when } e_1)\end{array}\right)$$

Proof: This follows from lemmas A.1 and A.2.

Theorem A.2
$R(p_1 \Rightarrow p_2 \textbf{ when } e_1 \mid e_2) = R(p_1 \Rightarrow p_2 \textbf{ when } e_1) \cup R(p_1 \Rightarrow p_2 \textbf{ when } e_2)$

Proof:

$$\begin{array}{l}
R(p_1 \Rightarrow p_2 \textbf{ when } e_1 \mid e_2) \\
= \{(v_1,v_2) \mid \theta(p_1) = v_1 \wedge \theta(p_2) = v_2 \wedge (e_1 \mid e_2)(v_1)\} \\
= \{(v_1,v_2) \mid \theta(p_1) = v_1 \wedge \theta(p_2) = v_2 \wedge (e_1(v_1)) \mid (e_2(v_1))\} \\
= \{(v_1,v_2) \mid \theta(p_1) = v_1 \wedge \theta(p_2) = v_2 \wedge e_1(v_1)\} \cup \\
\qquad \{(v_1,v_2) \mid \theta(p_1) = v_1 \wedge \theta(p_2) = v_2 \wedge e_2(v_1)\} \\
= R(p_1 \Rightarrow p_2 \textbf{ when } e_1) \cup R(p_1 \Rightarrow p_2 \textbf{ when } e_2)
\end{array}$$

Theorem A.3
$R(p_1 \Rightarrow p_2) = R(p_1 \Rightarrow p_2 \textbf{ when } e) \cup R(p_1 \Rightarrow p_2 \textbf{ when } \neg e)$

Proof: This follows from lemma A.1 and theorem A.2.

Theorem A.4
For any chain which is constructed using the second transition of B_2 there will be a chain which does not use this transition and which is shorter.

Proof: The discarded transition has a condition which prevents either 1 being moved onto 2 when this is possible or reducing the number of blocks covering 1 and 2. It is easily shown by induction on the length of the chains that for any chain using the second transition there will be a chain which simply omits that usage.

Theorem A.5
For any chain which is produced using B_5 there will be an equivalent chain produced using B_6 which is shorter.

Proof: In shortening the transition chains, the omitted transitions are exactly those which are performed by the shortest chains, so removing them has the effect of shortening the shortest chains further.

References

Baader, F. and Siekmann, J. (1994) Unification theory. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, ed. M. Gabbay Dov, C. J. Hogger and J. A. Robinson. Oxford University Press.

Beynon-Davies, P. (1993) *Knowledge Engineering for Information Systems*. McGraw-Hill.

Bird, R. S. (1987) *A Calculus of Functions for Program Derivation*. Oxford University Programming Research Group Monograph.

Brownston, L., Farrell, R., Kant, E. and Martin, N. (1985) *Programming Expert Systems in OPS5: An introduction to Rule-Based Programming*. Addison-Wesley.

Burstall, R. M. and Darlington, J. (1977) A Transformation System for Developing Recursive Programs. *J. ACM* 24 (1).

Charniak, E. *et al.* (1987) *Artificial Intelligence Programming*. Lawrence Erlbaum.

Clark, A. N. (1994) Pattern recognition of noisy sequences of behavioural events using functional combinators. *Computer Journal*, **37**(5).

Clocksin, W. F. and Mellish, C. S. (1984) *Programming in Prolog*. 2nd ed. Springer-Verlag.

Culbert, C., Riley, G. and Savely, R. T. (1987) Approaches to the verification of rule-based expert systems. In *1st Annual Workshop on Space Operations Automation and Robotics (SOAR 87)*, NASA Conf. Pub. 2491.

Goguen, J. A., Thatcher, J. W. and Wagner, E. G. (1976) An initial algebra approach to the specification, correctness and implementation of abstract data types. in *Current Trends in Programming Methodology IV*, Yeh, R. (ed.). Prentice Hall.

Harper, R., MacQueen, D. and Milner, R. (1986) *Standard ML*. Laboratory for Foundations of Computer Science Report Series ECS-LFCS-86-2.

Hudak, P. *et al.* (1992) The Haskell Report. *ACM SIGPLAN Notices*, **27**(5).

Jounanaud, J-P. and Dershowitz, N. (1990) Rewrite systems. In *Handbook of Theoretical Computer Science*, J. Van Leeuwen (ed.). Elsevier.

Krause, P. *et al.* (1993) Can we formally specify a medical decision support system? *IEEE Expert*, June.

Lucas, P. and Van Der Gaag, L. (1991) *Principles of Expert Systems*. Addison-Wesley.

Luger, G. F. and Stubblefield, W. A. (1989) *Artificial Intelligence and the Design of Expert Systems*. Benjamin Cummings.

Luqi, V. B. and Yehudai, A. (1993) Using Transformations in Specification-Based Prototyping. *IEEE Trans. on Software Engineering*, **19**(5).

Lopez, B., Meseguer, P. and Plaza, E. (1990) Knowledge based systems validation: a state of the art. *AI Communications*, **3**(2).

Major, F., Lapalme, G. and Cedergren, R. (1991) Domain generating functions for solving constraint satisfaction problems. *Journal of Functional Programming*, **1**(2).

Morgan, C. (1990) *Programming from Specifications*. Prentice Hall.

Nilsson, N. J. (1980) *Principles of Artificial Intelligence*. Springer-Verlag.

O'Keefe, R. M., Balci, O. and Smith, E. P. (1987) Validating expert system performance. *IEEE Expert*, Winter.

Partsch, H. and Steinbruggen, R. (1983) Program transformation systems *ACM Computing Surveys*, **15**(3).

Peyton Jones, S. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall.

Plotkin, G. D. (1975) Call-by-name, Call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**.

Roman, G., Gamble, R. F. and Ball, W. E. (1993) Formal derivation of rule-based programs. *IEEE Trans. Software Engineering*, **19**(3).

Rushby, J. (1988) Quality Measures and Assurance for AI Software. NASA Contractor Report CR-4187.

Sondergaard, H. and Setsoft, P. (1992) Non-determinism in functional Languages. *The Computer Journal*, **35**(5).

Steele, G. (1990) *Common Lisp, the Language.* Digital Press.

Wadler, P. (1990) Comprehending monads. In *Proc. 19th Symp. on Lisp and Functional Programming*, Nice, France. ACM.