stichting
mathematisch
centrum

MC

AFDELING INFORMATICA (DEPARTMENT OF COMPUTER SCIENCE) IW 151/80 OKTOBER

A. DE BRUIN

OPERATIONAL AND DENOTATIONAL SEMANTICS DESCRIBING THE MATCHING PROCESS IN SNOBOL4

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1980 Mathematics subject classification: 68B10

ACM-Computing Reviews-category: 5.24

Operational and denotational semantics describing the matching process in SNOBOL4*)

by

A. de Bruin

ABSTRACT

The pattern matching process in SNOBOL4 is investigated. We consider a subset of the language which is simple in this respect that patterns are not allowed as values of variables. This leads to matching processes that always terminate. After an informal description of the matching algorithm we present an operational semantics in the SECD-machine style. This semantics uses a stack to implement the backtracking which can occur during matching. After that a denotational semantics is introduced which uses continuations to describe the backtracking. Equivalence of the two semantics is then proved. The operational and denotational semantics are as similar as possible while retaining the typical operational respectively denotational ideas. This leads to a straightforward equivalence proof. That this similarity pays off is shown by another, somewhat disparate operational semantics for which equivalence with the denotational semantics is much harder to prove.

KEY WORDS & PHRASES: *Denotational semantics, operational semantics, continuation, backtracking, pattern matching, SNOBOL4, SECD machines*

---

*) This report will be submitted for publication elsewhere.

1. INFORMAL DESCRIPTION OF THE SNOBOL4 FRAGMENT

This chapter serves as an introduction for those who are not acquainted with SNOBOL4, or only superficially so. We will discuss here only that part of the language which will be dealt with in the rest of the paper. Details and differences with full SNOBOL4 will be discussed in later chapters. The reader who is familiar with the language can therefore skip this chapter.

Pattern matching is in essence investigating whether a string of symbols (the *subject string*) is of a certain form as specified by a *pattern*. Another entity which has a role in this process is the *cursor*, a variable with integer values between zero and the length of the subject string, which serves as a pointer into this string. Cursor = 0 means that the pointer is located at the beginning of the string, the cursor being equal to the length of the subject string corresponds to the pointer being placed at the right-hand end of the string. In general, cursor = n denotes that the pointer is positioned between the n-th and the $(n+1)$-th symbol in the subject string.

Matching a string h against a pattern p starting with cursor position n may have two outcomes. The match may fail, which means that the substring of h starting directly after the n-th symbol does not have the form prescribed by p. The alternative is that the match succeeds in which case the process will yield a new cursor value, for instance n'. This will happen if the substring of h between cursor position n and n' has the property specified by p.

We will now present several forms which patterns can take, and explain their meaning. The first possibility is that a pattern is a string, for instance h'. Matching the subject h against the pattern h' succeeds if $h \equiv h_1 h' h_2$ and if the cursor is located just before h'. After the match the cursor will then be placed just before the substring h_2 in the subject. In all other cases the match will fail. For example, if the subject string is 'arie' and the pattern is 'ri', then the match fails if the precursor position (the value of the cursor before the match) equals 0, and the match will succeed if the precursor position is 1. In the latter case the corresponding postcursor value will be 3.

The pattern $\underline{nil}$ matches every string without altering the cursor position. Matching against this pattern is the same as matching against the pattern formed by the empty string. The pattern _fail_ is a pattern which fails invariably whatever the subject and the cursor position might be.

There are two dyadic operators on patterns, namely $\vee$ and &. The pattern $p_1 \vee p_2$ matches every string that matches p_1 or p_2, and $p_1 \& p_2$ is a pattern that matches every string matched by p_1 followed by any string that is matched by p_2. We have to be more precise here, that is we must describe in more detail how the *scanner* (the matching algorithm) works.

The patterns defined up till now consist of elementary subpatterns (strings *h*, _nil_ and _fail_) connected by &- and $\vee$-operators. In a pattern of the form $p_1 \& p_2$, p_2 is called the *subsequent* of p_1, and in $p_1 \vee p_2$, p_2 is called the *alternative* of p_1. The scanner acts in the following way: if it has to match against a (sub-)pattern of the form $p_1 \vee p_2$, then the scanner behaves first as if it has encountered only the pattern p_1. But the alternative p_2, together with the current cursor position, will be remembered in case the match will fail later on.

If the matching process fails at a certain instant then something takes place which is called *backtracking*. The scanner returns to the last "choice point" in the pattern which is the last encountered subpattern of the form $p_1 \vee p_2$, where p_2 has not yet been tried. It restores the situation (cursor position) to how it was just before choosing p_1 as first alternative. It now chooses p_2 and the matching process proceeds as if p_2 had been substituted for $p_1 \vee p_2$ in the pattern.

If the scanner hits upon a pattern of the form $p_1 \& p_2$ then it first tries to match against p_1. If this fails then the backtracking process as described above will take place. If it succeeds then the scanner tries to match the subject against p_2 starting with a new cursor value which is the result of the match against p_1. If the scanner fails and all alternatives are exhausted which means that no more backtracking is possible, then the whole match fails. The overall match succeeds if we come to the right-hand side of the pattern.

EXAMPLE. We try to match the string 'arie' against the pattern

('a' & ('ri' $\vee$ 'r')) & 'i'.

1. Cursor position = 0. Match against 'a' succeeds. The new cursor position

is 1 and we try to match against the subsequent ('ri' ∨ 'r') & 'i'.

2. This pattern has the form $p_1 \& p_2$, so we first try to match against the first component 'ri' ∨ 'r'.
3. This pattern has the form $p_1 \vee p_2$. We now act as if we have encountered the pattern 'ri' alone. We remember however the situation as it is now in order to be able to backtrack to it later on.
4. Match against 'ri' succeeds, and the cursor value is now 3. We next try to match against the subsequent, the second component from step 2.
5. This is the pattern 'i'. Match against 'i' fails and thus we have to backtrack.
6. The situation of step 3 is restored by setting the cursor position to 1 again and we try the alternative 'r' instead of 'ri'.
7. Match against 'r' succeeds. The new cursor position is 2. We try the subsequent, the second component of ('ri' ∨ 'r') & 'i'.
8. Match against this pattern 'i' succeeds, the new cursor position is 3.
9. There are no subsequents left and the whole match has thus succeeded.

Now that we know how the scanner works, we can describe the effect of the pattern *abort*. If the scanner encounters this pattern the whole matching process terminates. Notice the difference with *fail* which would force the scanner to backtrack.

We next discuss the way variables can be handled in the matching process. The variables will have strings as values. A variable *v* can be a pattern by itself. Such a pattern has the same meaning as the pattern *h*, where *h* is the value of *v* at the time the pattern expression is evaluated, which in general will be just before the match starts. There is an exception to this which will be discussed later on.

There are two ways to change values of variables during the match, namely by immediate and conditional assignment. Immediate assignment is indicated by patterns of the form *p$v*. The meaning of such a pattern is the following. If the scanner manages to match *p* against a substring of the subject, then this substring will be assigned to the variable *v*. This assignment is performed immediately, and always, even if the match will fail later on. Therefore a subpattern *p$v* can cause the variable *v* to be changed more than once in one match. For instance, during the match of the

the string 'arie' against the pattern (('a' $\vee$ 'ar') \$ *v*) & 'ie' the variable *v* will change value twice. First the string 'a' will be assigned to it and later the string 'ar'.

Conditional assignment is indicated by patterns of the form *p.v*, and has a similar effect on *v* as immediate assignment, apart from the fact that the assignment will be performed only if the local match against *p* was part of a full match which led to success. Only after the match has terminated successfully, the corresponding assignments will be performed.

EXAMPLES. If 'arie' is matched against ('a'.*v* $\vee$ 'ar') & 'ie' then *v* will not get another value; if we match the same string against ('a' $\vee$ 'ar'.*v*) & 'ie' then after the match *v* will have the value 'ar'.

As we remarked above, evaluation of a pattern expression, which is in essence replacing variables by their values, will in general take place before the match. We now give the exception hinted at earlier, which is the *unevaluated expression *p*. This pattern behaves in the same way as *p* does except for the fact that *p* will be evaluated at the moment the scanner encounters **p* during the match (which therefore can happen more than once).

EXAMPLE. The pattern ('a'\$*v* $\vee$ 'b'\$*v*) & (**v*) matches the strings 'aa' and 'bb'. This can be contrasted with the pattern ('a'\$*v* $\vee$ 'b'\$*v*) & *v* which matches 'a' followed by *h* or 'b' followed by *h*, where *h* is the value of *v* before the match. Notice also that this latter pattern has the same effect as ('a'.*v* $\vee$ 'b'.*v*) & *v* or ('a'.*v* $\vee$ 'b'.*v*) & (**v*).

In the first example ('a'\$*v* $\vee$ 'b'\$*v*) & (**v*) we see a typical example of the use of the *-operator. We combined it there with the \$-operator and were thus able to use the outcome of the match against the first component of the pattern while matching against the second component.

This concludes our informal discussion of the meaning of the patterns. The sequel of this paper will be devoted to more formal definitions of the process described above.

2. SYNTAX

We will now describe the syntax of the SNOBOL fragment, namely the pattern expressions that we allow in our language. We introduce the following syntactic classes, together with letters that denote typical elements of these classes.

$h \in \mathcal{Str}$, the *strings*. String values will be enclosed in quotation marks, for instance 'arie' denotes the string consisting of letters a, r, i and e consecutively. We denote the empty string by ''. If $h \in \mathcal{Str}$ and n,n' are integers such that $0 \le n \le n' \le k$, where k is the length of h, then $h[n:n']$ denotes the substring of h which begins with the $(n+1)$-th symbol and ends right after the n'-th symbol. If $0 \le n \le n' \le k$ does not hold then $h[n:n']$ denotes the empty string.

$v \in \mathcal{Var}$, the *variables*. In contrast to full SNOBOL4, we define explicitly a class of variables which is distinct from the class of strings.

$n \in \mathcal{Num}$, the *numerals* denoting nonnegative integers. We will use the letter n also to denote nonnegative integers themselves. No confusion will arise from this as the intended meaning can always be deduced from the context. We assume the existence of a *derepresentation function* $\mathcal{D}\colon \mathcal{Num} \to \mathrm{N}$, mapping numerals to the corresponding nonnegative integers, which will be injective and surjective. This means that every integer has exactly one numeral representing it and therefore $\mathcal{D}^{-1}$ is well defined. This heavy machinery might seem somewhat overdone, and in fact we could, for the moment, do without it and proceed a little less formally. However we maintain this function here because in a later stage it is needed anyway, and also other $\mathcal{D}$-functions have to be introduced (see chapter 5, definition 5.1).

$p \in \mathcal{Pat}$, the *patterns*. We give the following BNF-like definition.

$p ::= h \mid$	literal string
$v \mid$	variable
$\underline{nil} \mid \underline{abort} \mid \underline{fail}$	constants
$p\$v \mid$	immediate assignment
$p.v \mid$	conditional assignment

$n\$\$v \mid n..v \mid$	auxiliary patterns, not occurring in programs; needed in the operational semantics to describe the effect of immediate and conditional assignment
$*p \mid$	unevaluated expression
$p_1 \vee p_2 \mid$	alternation; we donot use the |-sign because the BNF notation does not allow this
$p_1 \& p_2$	concatenation; we chose the &-symbol instead of the space for the sake of clarity.

As we intend to study the matching process only we do not present the many other SNOBOL4 features. We also made a selection from the pattern structures which are possible in SNOBOL4. We have chosen the subset such that the essential aspects of the pattern matching process can be studied through it.

3. OPERATIONAL SEMANTICS

The semantics given here is inspired by a description of a SNOBOL4 implementation by GIMPEL [3]. There are some differences however.

The first one is that we allow only strings as values of variables. This has been done in order not to be forced to get into detail concerning coercion problems. Furthermore we do not include patterns as values because that would complicate the presentation a great deal, as we have to resort to recursively defined domains in the denotational semantics. This will be the subject of another paper.

Gimpel describes three phases in the elaboration of patterns. He talks about *compilation* which transforms a pattern expression into a tree representing it, *pattern building* which takes this tree, replaces variables by their values at that moment and builds a *pattern structure* (a graph representing the pattern tree in such a way that the scanner can traverse it efficiently), and *pattern matching*.

We do not distinguish these phases. Pattern match will be done directly from the pattern expression. We do not replace variables by their values but we add to the patterns a store *s* giving the values of the variables at pattern building time. In that way the scanner will be able, during the match, to find the meaning of a pattern component *v* by inspecting *s*.

The way the scanner performs the matching process, as described by Gimpel, is roughly as follows. Besides the cursor variable the scanner uses also the pattern structure which has been constructed by the pattern building process, and a variable, which we call *ptn* which has as a value a pointer into this structure indicating how far the match has proceeded in this structure. Furthermore there is a stack to save untried alternatives. The scanner now repeats the following loop.

1. If the pattern component pointed at by *ptn* has an alternative then save this alternative, represented by its *ptn* value, and the current cursor value on the stack.
2. Try to match the pattern component determined by the value of *ptn* (which is a primitive pattern: a string *h*, <u>*nil*</u>, <u>*fail*</u>, <u>*abort*</u>) against the subject string. If this succeeds goto step 3, else goto step 4.
3. Find out whether the pattern component designated by *ptn* has a subsequent. If so, set *ptn* to point at it and go back to step 1; if not, we are ready, and the overall match has succeeded.
4. (Backtrack step). Inspect the stack. If it is empty then we are done, there are no alternatives left, and the pattern match has failed. If the stack is not empty then pop the stack to find a new *ptn* value and a new cursor value. Assign these values and go back to step 1.

For those who are acquainted with SNOBOL4: the above algorithm describes the anchored fullscan mode, which will be the only mode to be dealt with in this paper.

Here the matching process will be defined in terms of an abstract machine, not unlike LANDIN's SECD machine [4]. We will give a function called *step* which performs the elementary steps of the process, changing the machine configuration, which we will now describe informally.

A machine configuration *m* is an 8-tuple $<p,s',h,c,a,q,n,s>$, where *p* is a pattern and *s'*,*s* are *stores* (lists of variable-value pairs). The store *s'* records the values of the variables as they were at pattern building time. The pair $<p,s'>$ determines a pattern structure, which corresponds roughly to the pattern component pointed at by the variable *ptn* in Gimpel's algorithm. Furthermore, h is the subject string, and c is the so called *subsequent* which is in essence a list of pattern components to be matched against once the match against p has terminated successfully. To be more precise, this is organized as follows: c is either equal to the list <READY>, the endmarker of the list, or it has the form $<p,s,h,c>$ where the pair $<p,s>$ determines the first pattern-structure component in the list, while c constitutes the tail of the list (the subject string h is included only for convenience). The item q in the 8-tuple is a list of variable-value pairs recording the conditional assignments encountered so far which have to be performed if the total match succeeds. Furthermore n is the numeral giving the present cursor value and s is the present store. Finally, the component a is the *alternative* which corresponds to the stack in Gimpel's description. This a is either equal to <FAIL> which denotes the bottom of the stack, or it is a 7-tuple of the form $<p,s,h,c,a,q,n>$. Here p, s and c correspond to the *ptn* value on the stack as given by Gimpel (p, s and c represent a point in the pattern structure: p and s determine a pattern structure component and c determines the subsequents of this component), n corresponds to the cursor value on Gimpel's stack, q denotes the queue of conditional assignments accumulated up to the moment that particular stack frame was constructed (this has no analogon in Gimpel's stack because he uses a trick to circumvent this space consuming method), and finally a stands for the other frames of the stack.

The function *step* takes a machine configuration $m = <p,s',h,c,a,q,n,s>$, inspects the form of p and changes m correspondingly into a new configuration. We next present the formal definitions.

First some notational conventions. We will frequently use Curried functions which are functions that can take functions as arguments and yield functions as values. This generally leads to expressions with too many parentheses to be readable. To avoid this we leave parentheses out as much as possible using the convention that function application associates to the left. This means that *fabc* should be taken as $((f(a))(b))(c)$.

Function domain parentheses will be omitted under the convention that the $\rightarrow$- operator associates to the right. That is, $A \rightarrow B \rightarrow C$ should be read as $A \rightarrow (B \rightarrow C)$.

Concatenation of two lists l_1 and l_2 will be written as $l_1{}^\wedge l_2$. The empty list will be denoted by <>.

We define the following classes.

$s \in$ S, the *stores*. These are finite, and possibly empty, lists of elements from $\mathit{Var} \times \mathit{Str}$ ($<v,h>$-pairs). Elements from S are data structures which determine the values of the variables. Only the variables which have nonempty strings as values are recorded in a store s. In accordance with the convention in SNOBOL4 that all variables are initialized on the empty string there will be at any moment during program execution only a finite number of variables with nonempty values. We define two operations on stores.

1. updating. The store resulting from s by assigning the string h to v is represented by the list $s^\wedge<v,h>$.
2. extracting. The value of v in store s is denoted by $s⦇v⦈$. This is defined as follows.

a) $<>⦇v⦈ = ''$

b) $$(s^\wedge<v,h>)⦇w⦈ = \begin{cases} h & \text{if } v \equiv w, \\ s⦇w⦈ & \text{otherwise.} \end{cases}$$

Notice that more than one pair with first element v can occur in a store s. Only the rightmost pair "counts" however.

$q \in$ Q, the *queues of accumulated conditional assignments*. These too are finite and possibly empty lists of $<v,h>$-pairs. They constitute the queue of conditional assignments which have to be performed if the overall match succeeds. To accomplish this we simply concatenate the two lists: the store s' resulting from performing the assignments given by q in store s is given by $s' = s^\wedge q$

$c \in$ C, the *subsequents*. The class C is inductively defined as follows. An element c from C is either the list <READY> containing one element, or it is a list of the form $<p,s,h,c>$ where $p \in \mathit{Pat}$, $h \in \mathit{Str}$, $s \in$ S and c is again a subsequent.

$a \in A$, the *alternatives*. The class A is inductively defined by: an element a from A is either the one element list <FAIL> or a list $c^\wedge\langle a,q,n\rangle$ formed by concatenating a subsequent with a list containing an alternative, a $q \in Q$ and a numeral n.

$r \in R$, the *results*, i.e. the possible outcomes of a match. The class R is defined by $R = (\mathit{Num} \cup \{\text{FAIL},\text{ABORT}\}) \times S$. A results r is thus a pair $\langle \bar{n},s\rangle$ where $\bar{n}$ denotes the final cursor position (if the match was successful) and s is the resulting store.

$m \in M$, the *machine configurations*. A machine configuration is either a a final configuration which is an element from R, or an 8-tuple $a^\wedge\langle s\rangle$ formed by concatenating an alternative with a list containing a store s as only element. The predicate *final*(m) holds iff m is a final configuration.

We now have enough tools to define the step function.

The function *step*: $M \to M$ is defined as follows.

A. If m is a final configuration then $\mathit{step}(m) = m$.

B. If m has the form $\langle \text{READY},a,q,n,s\rangle$ then $\mathit{step}(m) = \langle n,s^\wedge q\rangle$.

C. In all other cases m has the form $\langle p,s',h,c,a,q,n,s\rangle$, and the definition proceeds by induction on the structue of p.

1. $\mathit{step}\langle h',s',h,c,a,q,n,s\rangle = \begin{cases} c^\wedge\langle a,q,\mathcal{D}^{-1}n'\rangle & \text{if } h' \equiv h[\mathcal{D}n{:}n'] \\ a^\wedge\langle s\rangle & \text{otherwise} \end{cases}$
2. $\mathit{step}\langle v,s',h,c,a,q,n,s\rangle = \langle s'(v),s',h,c,a,q,n,s\rangle$
3. $\mathit{step}\langle \underline{nil},s',h,c,a,q,n,s\rangle = c^\wedge\langle a,q,n,s\rangle$
4. $\mathit{step}\langle \underline{abort},s',h,c,a,q,n,s\rangle = \langle \text{ABORT},s\rangle$
5. $\mathit{step}\langle \underline{fail},s',h,c,a,q,n,s\rangle = a^\wedge\langle s\rangle$
6. $\mathit{step}\langle p\$v,s',h,c,a,q,n,s\rangle = \langle p,s',h,\langle n\$\$v,s',h,c\rangle,a,q,n,s\rangle$
7. $\mathit{step}\langle p.v,s',h,c,a,q,n,s\rangle = \langle p,s',h,\langle n..v,s',h,c\rangle,a,q,n,s\rangle$
8. $\mathit{step}\langle n'\$\$v,s',h,c,a,q,n,s\rangle = c^\wedge\langle a,q,n,s^\wedge\langle v,h[\mathcal{D}n'{:}\mathcal{D}n]\rangle\rangle$
9. $\mathit{step}\langle n'..v,s',h,c,a,q,n,s\rangle = c^\wedge\langle a,q^\wedge\langle v,h[\mathcal{D}n'{:}\mathcal{D}n']\rangle,n,s\rangle$
10. $\mathit{step}\langle *p,s',h,c,a,q,n,s\rangle = \langle p,s,h,c,a,q,n,s\rangle$
11. $\mathit{step}\langle p_1 \vee p_2,s',h,c,a,q,n,s\rangle = \langle p_1,s',h,c,\langle p_2,s',h,c,a,q,n\rangle,q,n,s\rangle$
12. $\mathit{step}\langle p_1 \& p_2,s',h,c,a,q,n,s\rangle = \langle p_1,s',h,\langle p_2,s',h,c\rangle,a,q,n,s\rangle$.

EXPLANATION.

Ad B. If during the match we encounter the end of the subsequent list, then the match has clearly succeeded (compare step 3 in Gimpel's algorithm). The postcursor position then is *n*, the cursor position on encountering READY, and the final store is obtained by performing all assignments in *q* from left to right.

Ad C. 1. If the pattern component is a literal string *h'* then we have to find out whether *h'* matches the subject string with respect to the present cursor position. If so, we have to continue with the next pattern component and this is given by the subsequent *c*. Now the store *s* and the queue *q* have not changed. Also there are no new alternatives found in the meantime so the alternative is still given by *a*. The only entity which has changed is the cursor position which must be set to its new value. By adding the list $<a,q,\mathcal{D}^{-1}n',s>$ to *c* we thus obtain a new machine configuration which reflects the effects of the successful match.

If the match against *h'* fails then we have to backtrack, and we take one frame from the stack *a*. Finding out that the match fails does not affect the store, so we only have to add <*s*> to get the resulting machine configuration.

2. If the pattern component is a variable *v* then we have to inspect the store as it was at pattern building time to find the value of *v*. This store is given by *s'*. The resulting machine configuration is then obtained by replacing *v* by the literal string which is the value of *v* in *s'*.

6-8. The pattern *p$v* is handled as follows: *p$v* is rewritten as *p* & (*n$$v*). So first a match against *p* is attempted. If this succeeds then we have to assign to *v* the substring of the subject which has been matched, and that is precisely the effect of matching against *n$$v*. This match always succeeds and has the side effect that the substring from the subject between cursor value *n* (given by the pattern component *n$$v*) and the present cursor value is assigned to *v*. So *n$$v* serves to indicate that an assignment has to be done, and it also provides the cursor value at the beginning of the match against *p*.

7-9. Similar to 6-8, but now the matched substring of the subject is added to the queue *q*.

10. The effect of matching against $*p$ in store s is the same as matching against the pattern structure derived from p in store s. So the only thing to be done is to replace s' by s.

11-12. In these cases the pattern is decomposed and the second component is retained in the new alternative, resp. the new subsequent.

Now that we have a step function which gives one step results, we can define the function P which takes a machine configuration and yields the final result of the matching process, a configuration m for which $final(m)$ holds. The function P is obtained by repeating the function *step* until a final configuration has been reached. We can formalize this in two ways. The first one is straightforward:

$$P(m) = \begin{cases} m' & \text{if there exists a row } m_1,\dots,m_k \text{ such that } m = m_1, \\ & m_k = m',\ m_{i+1} = step(m_i)\ (i = 1,\dots,k-1),\ \neg final(m_i) \\ & (i = 1,\dots,k-1),\ final(m_k), \\ \bot & \text{otherwise.} \end{cases}$$

There is another definition possible which is neater, but uses fixed point theory. The function P can be defined recursively by

$$P(m) \Leftarrow final(m) \rightarrow m,\ P(step(m)),$$

or more precisely

$$P = \mu[\lambda\phi\cdot\lambda m\cdot final(m) \rightarrow m,\ \phi(step(m))],$$

where μ is the least fixed point operator (see for instance DE BAKKER [1], or STOY [7] who calls this operator *fix*).

In order for the latter definition to make sense we have to impose a cpo structure on the class M of machine configurations. This can be done by adding the element $\bot$ to M and making M a discrete cpo ($m_1 \sqsubseteq m_2$ iff $m_1 = \bot$ or $m_1 = m_2$). We also extend the definition of *step* by taking $step\ \bot = \bot$. It can be shown (in the standard way) that the operator $\lambda\phi\cdot\lambda m\cdot final(m) \rightarrow m,\ \phi(step(m))$ is continuous, and thus that the least fixed point exists.

That the two definitions are equivalent can be shown in a standard way (see for instance [1, paragraph 3.3]).

Finally we define the operational meaning function $\mathcal{O}$ which gives the outcome of the process of matching a string h against a pattern p with initial store s.

$$\mathcal{O}[\![p]\!]\ h\ s = \mathcal{P}\langle p,s,h,\langle \mathrm{READY}\rangle,\langle \mathrm{FAIL}\rangle,\langle\rangle,0,s\rangle.$$

Here we used the convention that syntactical objects occurring as an argument of a function are enclosed in $[\![\cdot]\!]$-type brackets to make the expression more readable.

4. DENOTATIONAL SEMANTICS

We now turn to a discussion of the denotational semantics of our SNOBOL4 fragment. Before we do so however, we first make a remark on the notation we will use. The semantical classes used in the denotational semantics will be different from the ones in the above chapter. For instance we defined the class S of stores by $S = (\mathit{Var} \times \mathit{Str})^*$, but now we will take the domain of the stores to be $S = \mathit{Var} \to \mathit{Str}$. This can be done because we do not work any more with finite representations, we can use infinitary mathematical objects such as functions in the denotational semantics.

We will however use the same symbols to denote corresponding semantical classes and their typical elements. So in this chapter we define S with typical element s by $s \in S = \mathit{Var} \to \mathit{Str}$. This usage will not cause confusion in this chapter because here we will be occupied only with denotational domains and values. If confusion can be possible we will use the so called *diacritical convention* (MILNE & STRACHEY [5]): elements in the denotational world will be decorated with an acute accent ´, and the operational domains and values with a grave accent `. According to this convention we then can write $\acute{S} = \mathit{Var} \to \mathit{Str}$ and $\grave{S} = (\mathit{Var} \times \mathit{Str})^*$. Notice that Var and Str, being syntactic domains are not decorated.

We return to the denotational semantics. The meaning of a pattern p can be described by the effects resulting from a match of an arbitrary string h against p. This match, if it succeeds, will affect the cursor value n (which is now an element of N, the nonnegative integers), the store s, and it might

also add new conditional assignments to the ones already accumulated, as given by q (q will now be an element of $\acute{S} \to \acute{S}$ and denote the store transformation which is the result of performing all conditional assignments).
The meaning of p will also be dependent on the store s' at pattern evaluation time, s' provides the values of the free variables in p, i.e. those variables that are not bound by a $*$-operator.

Using a meaning function N, the effect of matching the string h against pattern p evaluated in s', with initial situation given by conditional assignments q, cursor position n and store s, would then be given by the expression $N[\![p]\!]\ s'\ h\ q\ s$. The value of this expression could be a triple $<q',n',s''>$ giving the new q-, n- and s-values. This set up does not work however, and this can be seen most directly by studying the case that the match of h against p fails. For how should the effect of backtracking be described in this setting?

The problem becomes clearer if we take a look at the compositionality principle, a main idea behind the denotational style of defining. This principle says that the meaning of a compound expression should be composed from the meaning of its parts. For instance, the meaning $N[\![(p_1 \vee p_2)\ \&\ p_3]\!]$ should be given in terms of $N[\![p_1]\!]$, $N[\![p_2]\!]$ and $N[\![p_3]\!]$ only.

Now matching against p_3 can fail and cause backtracking, a jump in the pattern to p_2. However in determining the meaning $N[\![p_3]\!]$ we donot have the pattern text p_2 at our disposal anymore, as was the case in the operational semantics. The standard solution for this kind of problems around jumps in programs is to work with continuations.

The trick is that we give the function $N[\![p]\!]\ s'\ h$ an extra argument a, called the alternative which describes the result of backtracking from p. The effect of backtracking is: recover the situation to the state it was in at the latest choice point and proceed from there on *with the new store* s. This effect is captured by a function $a \in A = S \to R$, where $R = (N \cup \{\text{FAIL,ABORT}\}) \times S$. The alternative a takes a store as argument and delivers the result r of the rest of the matching process. In other words, an alternative a is a function that describes the pattern matching process starting from the moment that backtracking out of p occurs.

So we add an extra argument a, and we have that now, if backtracking takes place, $N[\![p]\!]\ s'\ h\ a\ q\ n\ s$ denotes the final result of the whole match.

But this must then be the case too should the match succeed. It is therefore needed to give $\mathcal{N}[\![p]\!]\ s'\ h$ yet another argument, a subsequent c, describing how the pattern matching process proceeds if matching against p has terminated successfully. The subsequent will yield a result r in R, which will be dependent on four arguments describing the situation after the local match has succeeded: the new store s'', the postcursor position n', the conditional assignments accumulated q' and a new alternative a' which is determined by the alternative we had before matching against p, updated with the possible alternatives found while matching against p which have not yet been tried. We thus arrive at a functionality $c \in \mathrm{C} = \mathrm{A} \rightarrow \mathrm{Q} \rightarrow \mathrm{N} \rightarrow \mathrm{S} \rightarrow \mathrm{R}$.

A subsequent can be viewed as a function determining how the match proceeds from a certain point in the pattern text. An alternative can be looked upon in the same way, but more information is available at the moment an alternative is constructed (i.e. while matching on encountering a choice point $p_1 \vee p_2$), namely the precursor position, the conditional assignments gathered so far and also the alternatives remaining if the match fails in the process after backtracking to the choice point. The difference between the two is clearly reflected in the respective functionalities $\mathrm{A} \rightarrow \mathrm{Q} \rightarrow \mathrm{N} \rightarrow \mathrm{S} \rightarrow \mathrm{R}$ vs. $\mathrm{S} \rightarrow \mathrm{R}$: an alternative is like a subsequent but not more dependent on a, q and n.

Concluding, the result $r = \mathcal{N}[\![p]\!]\ s'\ h\ c\ a\ q\ n\ s$ can be described as follows: $\mathcal{N}[\![p]\!]\ s'$ denotes the pattern structure resulting from evaluation of the expression p in store s'. Suppose h is matched against this pattern structure, and the initial situation is given by cursor position n, initial store s and conditional assignments accumulated so far determined by q. Suppose furthermore that the effect of the future of the matching process once match against p has been finished is given by a in case backtracking out of p occurs, and by c in case the match against p terminates successfully. In that case the final result of the whole matcing process is given by r.

The above discussion leads to the following definitions of domains and functionalities.

$s \in S = \mathit{Var} \to \mathit{Str}$	stores
$n \in N$	nonnegative integers
$q \in Q = S \to S$	accumulated conditional assignments
$r \in R = (N \cup \{\text{FAIL},\text{ABORT}\}) \times S$	results
$a \in A = S \to R$	alternatives
$c \in C = A \to Q \to N \to S \to R$	subsequents

We define a *variant* $s\{h/v\}$ of a store s by

$$(s\{h/v\})(w) = \begin{cases} s(w) & \text{if } v \not\equiv w \\ h & \text{if } v \equiv w. \end{cases}$$

Notice that the classes introduced above are not cpo's. Cpo's are not needed here because the semantic definition of N to come is a purely inductive one. No use is made of fixed points, and we also donot use recursively defined domains.

The semantic function N has functionality

$$N\colon \mathit{Pat} \to S \to \mathit{Str} \to C \to A \to Q \to N \to S \to R$$

and is defined by induction on the structure of its first argument as follows.

1. $N[\![h']\!]\ s'\ h\ c\ a\ q\ n\ s = \begin{cases} c\ a\ q\ n'\ s & \text{if } h' \equiv h[n:n'] \\ a\ s & \text{otherwise} \end{cases}$
2. $N[\![v]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![s'(v)]\!]\ s'\ h\ c\ a\ q\ n\ s.$
3. $N[\![\underline{nil}]\!]\ s'\ h\ c\ a\ q\ n\ s = c\ a\ q\ n\ s$
4. $N[\![\underline{abort}]\!]\ s'\ h\ c\ a\ q\ n\ s = \langle\text{ABORT},s\rangle$
5. $N[\![\underline{fail}]\!]\ s'\ h\ c\ a\ q\ n\ s = a\ s$
6. $N[\![p\$v]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p]\!]\ s'\ h\ c'\ a\ q\ n\ s$
   where $c' = \lambda a'\cdot\lambda q'\cdot\lambda n'\cdot\lambda s''\cdot c\ \ a'\ q'\ n'(s''\{h[n:n']/v\})$
7. $N[\![p\cdot v]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p]\!]\ s'\ h\ c'\ a\ q\ n\ s$
   where $c' = \lambda a'\cdot\lambda q'\cdot\lambda n'\cdot\lambda s''\cdot c\ a'(\lambda s\cdot(q's)\{h[n:n']/v\})n's''$
8. $N[\![\bar{n}\$\$v]\!]\ s'\ h\ c\ a\ q\ n\ s = c\ a\ q\ n(s\{h[\mathcal{D}\bar{n}:n]/v\})$
9. $N[\![\bar{n}..v]\!]\ s'\ h\ c\ a\ q\ n\ s = c\ a(\lambda s\cdot(q\ s)\{h[\mathcal{D}\bar{n}:n]/v\})\ n\ s$
10. $N[\![*p]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p]\!]\ s\ h\ c\ a\ q\ n\ s$
11. $N[\![p_1 \vee p_2]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p_1]\!]\ s'\ h\ c\{N[\![p_2]\!]\ s'\ h\ c\ a\ q\ n\}\ q\ n\ s$
12. $N[\![p_1 \& p_2]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p_1]\!]\ s'\ h\{N[\![p_2]\!]\ s'\ h\ c\}\ a\ q\ n\ s.$

REMARKS. Ad 1. If h matches h' at cursor position n then the remainder of the matching process is given by the subsequent c. The alternative, the conditional assignment queue and the store did not change, only the cursor has a new value. If h does not match then the remainder of the matching process is given by a which has to be applied to the current store s.

Ad 2. Like 1, but first the value of v in s' has to be determined. Notice that, in order to be able to maintain the above definition as one by structural induction, we have to choose a measure of complexity of patterns which guarantees that the complexity of variable is higher than that of a string. This can be accomplished easily though.

Ad 6. Matching gainst $p\$v$ is in principle the same as matching against p. Only when the match against p succeeds we have to aditionally assign the string matched to. This is taken care of by the new subsequent c' which describes the effect of this assignment followed by the effect of the old subsequent c.

Ad 7. Like 6, but now the new subsequent c' causes q to be updated instead of s.

Ad 8,9. Notice that the $\bar{n}$ occurring in the patterns is a numeral. Therefore $\bar{n}$ has to be changed into the corresponding number. Strictly speaking, clauses 8 and 9 are not needed in the definition, because patterns $\bar{n}\$\$v$, $\bar{n}..v$ do not occur in programs, nor in the right-hand sides of the other clauses in the definition. These auxiliary patterns have only been introduced for the sake of the operational definition, where the meaning of $p\$v$ has been defined in terms of the meaning of some $\bar{n}\$\$v$. See also the lemmas at the end of this chapter. We maintained these clauses here, because we will need them in proving the operational and denotational semantics equivalent.

Ad 11. Matching against $p_1 \vee p_2$ amounts to matching against p_1, but now we have a new alternative. On backtracking we have to match against p_2 in the situation as it is now (apart from the new store). This effect is taken care of by the new alternative $N[\![p_2]\!]\ s'\ h\ c\ a\ q\ n$. Notice that this alternative has the right functionality.

Ad 12. As in 11, but now a new subsequent is formed.

Finally we define the denotational counterpart of the function $\mathcal{O}$ from chapter 3. This is the semantic function M with functionality
$M\colon \mathit{Pat} \to \mathit{Str} \to S \to R$.

$$M[\![p]\!]\ h\ s = N[\![p]\!]\ s\ \ h\ \ \mathit{ready}\ \mathit{fail}\ \{\lambda s\cdot s\}\ 0\ s,$$

where the subsequent *ready* is defined by

$$\mathit{ready}\ a\ q\ n\ s = \langle n, q\ s\rangle$$

and the alternative *fail* by

$$\mathit{fail}\ s = \langle \mathrm{FAIL}, s\rangle.$$

So the complete matching of a string h against a pattern p in store s, corresponds to evaluating p in s, and matching h against this pattern structure. If this match succeeds then the accumulated conditional assignments have to be performed and this is handled by the subsequent *ready* which yields the postcursor position and the updated store. If the match fails then this has to be reported and that is what the alternative *fail* is for. Furthermore, the precursor position is 0, and the initial store is s. Finally, in the beginning there are no conditional assignments accumulated and this is denoted by the identity function $\lambda s\cdot s$.

We close this chapter by giving a lemma on the relation between clauses 6 and 8 (7 and 9) of the definition of N.

LEMMA 4.1.
1. $N[\![p\$v]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p\&(\bar{n}\$\$v)]\!]\ s'\ h\ c\ a\ q\ n\ s$
2. $N[\![p.v]\!]\ s'\ h\ c\ a\ q\ n\ s = N[\![p\&(\bar{n}..v)]\!]\ s'\ h\ c\ a\ q\ n\ s$

where $\bar{n} = \mathcal{D}^{-1}n$.

PROOF. The proof is straightforward by writing out the expressions. For instance in 1 we have: left-hand side $= N[\![p]\!]\ s'\ h\ c'\ a\ q\ n\ s$, where $c'\ a'\ q'\ n'\ s'' = c\ a'\ q'\ n'(s''\{h[n{:}n']/v\})$, and right-hand side $=$ $N[\![p]\!]\ s'\ h\{N[\![\bar{n}\$\$v]\!]s'\ h\ c\}\ a\ q\ n\ s$. So there remains to be proved $c' = N[\![\bar{n}\$\$v]\!]\ s'\ h\ c$, and this follows from the fact that $N[\![\bar{n}\$\$v]\!]\ s'\ h\ c\ a'\ q'\ n'\ s'' = c'\ a'\ q'\ n'(s''\{h[\mathcal{D}\bar{n}{:}n']/v\})$ and that $\mathcal{D}\bar{n} = \mathcal{D}\mathcal{D}^{-1}n = n$. □

5. OPERATIONAL AND DENOTATIONAL SEMANTICS ARE EQUIVALENT

Of course it is not by coincidence that the two semantics presented here are that similar. An example of the difficulties that one encounters if one chooses a more dissimilar pair of semantics will be given in the next chapter. Notice however, that there are essential differences between the two semantics. A first one is that the objects that are manipulated in the operational semantics are all finite representations (they are in fact BNF-definable) while the denotational semantics handles infinitary abstract objects. A more fundamental difference is that the denotational semantics is fully compositional while the operational semantics is not. Related to this is the fact that in the denotational semantics the outcome of the matching process is obtained, so to speak immediately, by applying the meaning function $\mathcal{M}$ to the suitable arguments, while in the operational semantics we get the result by letting an abstract machine compute it step by step.

If now we want co compare the two semantics the first thing to do is to find a correspondence between the operational domains and the denotational ones. The main theorem to be proved here is that the two functions $\mathcal{O}$ and $\mathcal{M}$, applied to corresponding arguments, will yield corresponding results. There is a straightforward correspondence between the domains, which will be given by the *derepresentation functions* $\mathcal{D}_X$ (one for every pair of domains $\acute{X}$ and $\grave{X}$) which map an element from the operational domain $\grave{X}$ onto the corresponding element in $\acute{X}$. So we will define functions $\mathcal{D}_S$, $\mathcal{D}_N$ (this is the function introduced already in chapter 2, which relates numerals and numbers), $\mathcal{D}_Q$, $\mathcal{D}_R$, $\mathcal{D}_A$ and $\mathcal{D}_C$. In the sequel we will use the convention that the subscripts will be omitted if this causes no confusion (this has already been done in chapter 2). We remark now already that these functions $\mathcal{D}$ will in general be neither one to one nor onto.

The fact to be proved in this chapter can now be stated as $\mathcal{D}(\mathcal{O}[\![p]\!]\ h\ s) = \mathcal{M}[\![p]\!]\ h\ (\mathcal{D}s)$. We will first give the definitions and afterwards provide some comments on these.

DEFINITION 5.1.

$\mathcal{D}_S\colon \grave{S} \to \acute{S}$ is defined by $\mathcal{D}_S\langle\rangle = \lambda v\cdot ""$

$$\mathcal{D}_S(\grave{s}\,{}^\wedge\langle v,h\rangle) = (\mathcal{D}_S\grave{s})\{h/v\}$$

$\mathcal{D}_Q\colon \grave{Q} \to \acute{Q}$ is defined by $\mathcal{D}_Q\langle\rangle = \lambda\acute{s}\cdot\acute{s}$

$$\mathcal{D}_Q(\grave{q}\,{}^\wedge\langle v,h\rangle) = \lambda\acute{s}\cdot((\mathcal{D}_Q\grave{q}\acute{s})\{h/v\})$$

$\mathcal{D}_R\colon \grave{R} \to \acute{R}$ is defined by $\mathcal{D}_R\langle\bar{n},\grave{s}\rangle = \langle\mathcal{D}_{\bar{N}}\bar{n},\mathcal{D}_S\grave{s}\rangle$

where $\mathcal{D}_{\bar{N}}\colon \mathit{Num} \cup \{\mathrm{FAIL},\mathrm{ABORT}\} \to \mathrm{N} \cup \{\mathrm{FAIL},\mathrm{ABORT}\}$

is defined by $\begin{cases} \mathcal{D}_N\bar{n} & \text{if } \bar{n} \in \mathit{Num} \\ \bar{n} & \text{if } \bar{n} \in \{\mathrm{FAIL},\mathrm{ABORT}\} \end{cases}$

$\mathcal{D}_C\colon \grave{C} \to \acute{C}$ is defined by $\mathcal{D}_C\langle\mathrm{READY}\rangle = \mathit{ready}$

$$\mathcal{D}_C\langle p,\grave{s},h,\grave{c}\rangle = \mathcal{N}[\![p]\!]\,(\mathcal{D}_S\grave{s})\;h\;(\mathcal{D}_C\grave{c})$$

$\mathcal{D}_A\colon \grave{A} \to \acute{A}$ is defined by $\mathcal{D}_A\langle\mathrm{FAIL}\rangle = \mathit{fail}$

$$\mathcal{D}_A(\grave{c}\,{}^\wedge\langle\grave{a},\grave{q},\grave{n}\rangle) = (\mathcal{D}_C\grave{c})\,(\mathcal{D}_A\grave{a})\,(\mathcal{D}_Q\grave{q})\,(\mathcal{D}_N\grave{n}).$$

REMARKS. We have $\mathcal{D}_S\colon (\mathit{Var} \times \mathit{Str})^* \to (\mathit{Var} \to \mathit{Str})$. Now the empty list corresponds to the situation that all variables have the empty string as value, so this accounts for the first line in the definition. Furthermore in the store $\grave{s}\,{}^\wedge\langle v,h\rangle$ all variables have the same value as in $\grave{s}$, except for v which has the value h, and this is reflected in the second line of the definition.

The function $\mathcal{D}_N$ has already been introduced in the second chapter. It has not been defined there, and we could not do so because we chose not to define the form of the elements in Num.

The functionality of $\mathcal{D}_Q$ is $(\mathit{Var} \times \mathit{Str})^* \to (\acute{S} \to \acute{S})$. The queue $\grave{q}$ in $\grave{Q}$ provides the conditional assignments to be performed from left to right. The corresponding function $\mathcal{D}_Q\grave{q}$ is the store transformation that describes the effect of performing these assignments. So we have $\mathcal{D}_Q\langle\rangle = \lambda\acute{s}\cdot\acute{s}$, for if the queue is empty then the store does not change. The second line of the definition can be phrased as follows: performing the assignments in the queue $\grave{q}\,{}^\wedge\langle v,h\rangle$ amounts to performing the assignments in $\grave{q}$ first and afterwards assigning h to v.

The function $\mathcal{D}_R$ is defined straightforwardly.

That $\mathcal{D}_C$<READY> should be equal to *ready* can be seen from the fact that they "do the same job": the accumulated conditional assignments are performed and the final result is delivered. This is formalized in Lemma 5.3. The next line in the definition can be commented upon as follows. The subsequent $<p,\tilde{s},h,\tilde{c}>$ describes a match of h against p evaluated in $\tilde{s}$, followed by subsequent $\tilde{c}$, and the entity $N[\![p]\!]\,(\mathcal{D}_S\tilde{s})\,h\,(\mathcal{D}_C\tilde{c})$ describes the same process for the corresponding elements in the denotational world.

Similar remarks as given on $\mathcal{D}_C$ apply for the function $\mathcal{D}_A$.

We next state some lemmas giving results on these functions.

<u>LEMMA 5.2</u>. $\mathcal{D}(\tilde{s}^\wedge\tilde{q}) = (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})$.

<u>PROOF</u>. Remind that the list $\tilde{s}^\wedge\tilde{q}$ represents the store resulting from performing the conditional assignments in $\tilde{q}$ on $\tilde{s}$ (see the definition of *step*<READY,...> in chapter 3). The proof is by induction on the length of $\tilde{q}$.

<u>Basis</u>. $\mathcal{D}(\tilde{s}^\wedge{<>}) = \mathcal{D}\tilde{s}$ and $(\mathcal{D}{<>})(\mathcal{D}\tilde{s}) = (\lambda\acute{s}\cdot\acute{s})(\mathcal{D}\tilde{s}) = \mathcal{D}\tilde{s}$.

<u>Induction step</u>. $\mathcal{D}(\tilde{s}^\wedge(\tilde{q}^\wedge{<v,h>})) = \mathcal{D}((\tilde{s}^\wedge\tilde{q})^\wedge{<v,h>}) =$
$(\mathcal{D}(\tilde{s}^\wedge\tilde{q}))\{h/v\}$.

On the other hand $(\mathcal{D}(\tilde{q}^\wedge{<v,h>}))(\mathcal{D}\tilde{s}) = [\lambda\acute{s}\cdot((\mathcal{D}\tilde{q}\acute{s})\{h/v\})](\mathcal{D}\tilde{s}) =$
$[(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})]\{h/v\}$,

and the result holds by induction. □

<u>LEMMA 5.3</u>. $\mathcal{D}(step{<}\text{READY},\tilde{a},\tilde{q},\tilde{n},\tilde{s}{>}) = ready(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

<u>PROOF</u>. The left-hand side is equal to $\mathcal{D}{<}\tilde{n},\tilde{s}^\wedge\tilde{q}{>} = {<}\mathcal{D}\tilde{n},\mathcal{D}(\tilde{s}^\wedge\tilde{q}){>}$, and the right-hand side equals ${<}\mathcal{D}\tilde{n},(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s}){>}$. The result now holds by the preceding lemma. □

<u>LEMMA 5.4</u>. $\mathcal{D}(P{<}\text{READY},\tilde{a},\tilde{q},\tilde{n},\tilde{s}{>}) = ready(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

<u>PROOF</u>. Because *step*<READY,$\tilde{a},\tilde{q},\tilde{n},\tilde{s}$> is final, we have that $P{<}\text{READY},\tilde{a},\tilde{q},\tilde{n},\tilde{s}{>}$ is equal to this, and the lemma immediately follows from Lemma 5.3. □

<u>LEMMA 5.5</u>. $\mathcal{D}{<}\text{FAIL},\tilde{s}{>} = fail(\mathcal{D}\tilde{s})$.

<u>PROOF</u>. Immediate by writing out the expressions. □

LEMMA 5.6. $\tilde{s}\llparenthesis v\rrparenthesis = (\mathcal{D}\tilde{s})(v)$.

PROOF. Induction on the length of $\tilde{s}$. The basic step is OK because $<>\llparenthesis v\rrparenthesis = ''$ and $(\mathcal{D}<>)(v) = (\lambda v\cdot '')(v) = ''$.

Induction step.

$$(\tilde{s}^\wedge<w,h>)\llparenthesis v\rrparenthesis = \begin{cases} h & \text{if } w \equiv v \\ \tilde{s}\llparenthesis v\rrparenthesis & \text{otherwise} \end{cases}$$

$$\mathcal{D}(\tilde{s}^\wedge<w,h>)(v) = ((\mathcal{D}\tilde{s})\{h/w\})(v) = \begin{cases} h & \text{if } w \equiv v \\ (\mathcal{D}\tilde{s})(v) & \text{otherwise.} \end{cases} \qquad \square$$

Now we want to prove $\mathcal{D}(\mathcal{O}[\![p]\!]h\ \tilde{s}) = \mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s})$. By writing out, using the definition of $\mathcal{O}$ and $\mathcal{M}$, this is equivalent to

$$\mathcal{D}(\mathcal{P}<p,\tilde{s},h,<\text{READY}>,<\text{FAIL}>,<>,0,\tilde{s}>) =$$
$$\mathcal{N}[\![p]\!]\ (\mathcal{D}\tilde{s})\ h\ \mathit{ready}\ \mathit{fail}\ (\lambda\tilde{s}\cdot\tilde{s})\ 0\ (\mathcal{D}\tilde{s}).$$

We distinghuish two cases, namely that the left-hand side of the above equality is unequal to $\perp$ and the case that it is equal to $\perp$. We establish the desired result for the first case by proving the following more general result.

LEMMA 5.7. For all $\tilde{c},\tilde{a},\tilde{q},\tilde{n}$ and $\tilde{s}$ we have: if $\mathcal{P}(\tilde{c}^\wedge<\tilde{a},\tilde{q},\tilde{n},\tilde{s}>) \neq \perp$ then $\mathcal{D}(\mathcal{P}(\tilde{c}^\wedge<\tilde{a},\tilde{q},\tilde{n},\tilde{s}>)) = (\mathcal{D}\tilde{c})(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

PROOF. The proof is by induction, essentially on the length of the computation. Now we have given two definitions of $\mathcal{P}$, and the induction argument depends on the definition chosen. If one thinks in terms of the fixed point definition then we have to use Scott's induction (fixed point induction), that is we have to prove that the lemma holds for $\lambda m\cdot\perp$ instead of $\mathcal{P}$ (which is clearly true), and that the lemma holds for $\lambda m\cdot \mathit{final}(m) \rightarrow m,\ \Phi(\mathit{step}(m))$ given that the lemma holds for the function Φ instead of $\mathcal{P}$. The proof given below can be reorganized in these terms.

If one adopts the other definition using rows of machine configurations, then the induction is simply on the length of the row. The basic step is again vacuously fulfilled because one easily sees that there are no zero step reductions from $\grave{c}^\wedge\langle\grave{a},\grave{q},\grave{n},\grave{s}\rangle$ since this configuration is not final. For the induction steps we distinguish thirteen cases, dependent on the form of $\grave{c}$.

1. $\grave{c} = \langle\text{READY}\rangle$. The lemma holds by Lemma 5.4.

2. $\grave{c} = \langle h',\grave{s}_1,h,\grave{c}_1\rangle$. There are two cases:
   a. $h[\mathcal{D}\grave{n}:\bar{n}] \equiv h'$ for some $\bar{n}$. We then have to prove
   $\mathcal{D}(\mathcal{P}(\grave{c}_1{}^\wedge\langle\grave{a},\grave{q},\mathcal{D}^{-1}\bar{n},\grave{s}\rangle)) = (\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})\ \bar{n}\ (\mathcal{D}\grave{s})$
   and this holds by induction and the fact that $\mathcal{D}\ \mathcal{D}^{-1}\ \bar{n} = \bar{n}$.
   b. Otherwise. Then the property to be proven is equivalent to
   $\mathcal{D}(\mathcal{P}(\grave{a}^\wedge\langle\grave{s}\rangle)) = (\mathcal{D}\grave{a})(\mathcal{D}\grave{s})$. Again there are two cases:
      I. $\grave{a} = \langle\text{FAIL}\rangle$. In this case we have to prove
      $\mathcal{D}(\mathcal{P}\langle\text{FAIL},\grave{s}\rangle) = \mathit{fail}(\mathcal{D}\grave{s})$ and this is an immediate consequence of Lemma 5.5.
      II. $\grave{a} = \grave{c}_2{}^\wedge\langle\grave{a}_1,\grave{q}_1,\grave{n}_1\rangle$. Then we have to prove
      $\mathcal{D}(\mathcal{P}\langle\grave{c}_2,\grave{a}_1,\grave{q}_1,\grave{n}_1,\grave{s}\rangle) = (\mathcal{D}\grave{c}_2)(\mathcal{D}\grave{a}_1)(\mathcal{D}\grave{q}_1)(\mathcal{D}\grave{n}_1)(\mathcal{D}\grave{s})$
      and this holds by induction.

3. $\grave{c} = \langle v,\grave{s}_1,h,\grave{c}_1\rangle$. We have to prove
   $\mathcal{D}(\mathcal{P}\langle\grave{s}_1\llparenthesis v\rrparenthesis,\grave{s}_1,h,\grave{c}_1,\grave{a},\grave{q},\grave{n},\grave{s}\rangle) =$
   $\mathcal{N}\llbracket v\rrbracket(\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$
   $\mathcal{N}\llbracket(\mathcal{D}\grave{s}_1)(v)\rrbracket(\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) = (\#)$
   We have by Lemma 5.6 and the definition of $\mathcal{D}_C$ that
   $(\#) = (\mathcal{D}\langle\grave{s}_1\llparenthesis v\rrparenthesis,\grave{s}_1,h,\grave{c}_1\rangle)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$,
   and now we can apply the induction hypothesis.

4. $\grave{c} = \langle\underline{nil},\grave{s}_1,h,\grave{c}_1\rangle$. Like 2a.

5. $\grave{c} = \langle\underline{abort},\grave{s}_1,h,\grave{c}_1\rangle$. Immediate.

6. $\grave{c} = \langle\underline{fail},\grave{s}_1,h,\grave{c}_1\rangle$. Like 2b.

7. $\grave{c} = \langle p\$v,\grave{s}_1,h,\grave{c}_1\rangle$. We have to prove
   $\mathcal{D}(\mathcal{P}\langle p,\grave{s}_1,h,\langle\grave{n}\$\$v,\grave{s}_1,h,\grave{c}_1\rangle,\grave{a},\grave{q},\grave{n},\grave{s}\rangle) =$
   $\quad\mathcal{N}\llbracket p\$v\rrbracket(\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$.
   We have
   $\mathcal{D}(\mathcal{P}\langle p,\grave{s}_1,h,\langle\grave{n}\$\$v,\grave{s}_1,h,\grave{c}_1\rangle,\grave{a},\grave{q},\grave{n},\grave{s}\rangle) =$ (ind.hyp.)
   $(\mathcal{D}\langle p,\grave{s}_1,h,\langle\grave{n}\$\$v,\grave{s}_1,h,\grave{c}_1\rangle\rangle)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (def. $\mathcal{D}_C$)
   $(\mathcal{N}\llbracket p\rrbracket(\mathcal{D}\grave{s}_1)h\{\mathcal{N}\llbracket\grave{n}\$\$v\rrbracket(\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)\})(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (def. $\mathcal{N}$)
   $\mathcal{N}\llbracket p\&(\grave{n}\$\$v)\rrbracket(\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$

$\mathcal{N}[\![p\$v]\!](\mathcal{D}\tilde{s}_1)h(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

The last equality holds by Lemma 4.1 and the fact that $\mathcal{D}^{-1}\mathcal{D}\tilde{n} = \tilde{n}$.

8. $\tilde{c} = \langle p.v,\tilde{s}_1,h,\tilde{c}_1\rangle$. Analogous to 7.

9. $\tilde{c} = \langle \tilde{n}_1\$\$v,\tilde{s}_1,h,\tilde{c}_1\rangle$. We have to prove
$\mathcal{D}(\mathcal{P}(\tilde{c}_1{}^\wedge\langle\tilde{a},\tilde{q},\tilde{n},\tilde{s}^\wedge\langle v,h[\mathcal{D}\tilde{n}_1:\mathcal{D}\tilde{n}]\rangle\rangle)) =$
$(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})((\mathcal{D}\tilde{s})\{h[\mathcal{D}\tilde{n}_1:\mathcal{D}\tilde{n}]/v\})$
and this holds by induction and the definition of $\mathcal{D}_S$.

10. $\tilde{c} = \langle \tilde{n}_1..v,\tilde{s}_1,h,\tilde{c}_1\rangle$. We have to prove
$\mathcal{D}(\mathcal{P}(\tilde{c}_1{}^\wedge\langle\tilde{a},\tilde{q}^\wedge\langle v,h[\mathcal{D}\tilde{n}_1:\mathcal{D}\tilde{n}]\rangle,\tilde{n},\tilde{s}\rangle)) =$
$(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\lambda\tilde{s}\cdot(\mathcal{D}\tilde{q}\tilde{s})\{h[\mathcal{D}\tilde{n}_1:\mathcal{D}\tilde{n}]/v\})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$
and this holds by induction and the definition of $\mathcal{D}_Q$.

11. $\tilde{c} = \langle *p,\tilde{s}_1,h,\tilde{c}_1\rangle$. We have to prove
$\mathcal{D}(\mathcal{P}\langle p,\tilde{s},h,\tilde{c}_1,\tilde{a},\tilde{q},\tilde{n},\tilde{s}\rangle) = \mathcal{N}[\![p]\!](\mathcal{D}\tilde{s})h(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$,
which holds by induction and the definition of $\mathcal{D}_C$.

12. $\tilde{c} = \langle p_1\vee p_2,\tilde{s}_1,h,\tilde{c}_1\rangle$. We have
$\mathcal{D}(\mathcal{P}(\tilde{c}^\wedge\langle\tilde{a},\tilde{q},\tilde{n},\tilde{s}\rangle)) =$
$\mathcal{D}(\mathcal{P}\langle p_1,\tilde{s}_1,h,\tilde{c}_1,\langle p_2,\tilde{s}_1,h,\tilde{c}_1,\tilde{a},\tilde{q},\tilde{n}\rangle,\tilde{q},\tilde{n},\tilde{s}\rangle) =$ (ind. hyp.)
$(\mathcal{D}\langle p_1,\tilde{s}_1,h,\tilde{c}_1\rangle)(\mathcal{D}\langle p_2,\tilde{s}_1,h,\tilde{c}_1,\tilde{a},\tilde{q},\tilde{n}\rangle)(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$ (def. $\mathcal{D}_C,\mathcal{D}_A$)
$\mathcal{N}[\![p_1]\!](\mathcal{D}\tilde{s}_1)h(\mathcal{D}\tilde{c}_1)\{\mathcal{N}[\![p_2]\!](\mathcal{D}\tilde{s}_1)h(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})\}(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$
$\mathcal{N}[\![p_1\vee p_2]\!](\mathcal{D}\tilde{s}_1)h(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$ (def. $\mathcal{D}_C$)
$(\mathcal{D}\langle p_1\vee p_2,\tilde{s}_1,h,\tilde{c}_1\rangle)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

13. $\tilde{c} = \langle p_1\&p_2,\tilde{s}_1,h,\tilde{c}_1\rangle$. We have
$\mathcal{D}(\mathcal{P}(\tilde{c}^\wedge\langle\tilde{a},\tilde{q},\tilde{n},\tilde{s}\rangle)) =$
$\mathcal{D}(\mathcal{P}\langle p_1,\tilde{s}_1,h,\langle p_2,\tilde{s}_1,h,\tilde{c}_1\rangle,\tilde{a},\tilde{q},\tilde{n},\tilde{s}\rangle) =$ (ind. hyp.)
$(\mathcal{D}\langle p_1,\tilde{s}_1,h,\langle p_2,\tilde{s}_1,h,\tilde{c}_1\rangle\rangle)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$ (2× def. $\mathcal{D}_C$)
$\mathcal{N}[\![p_1]\!](\mathcal{D}\tilde{s}_1)\ h\{\mathcal{N}[\![p_2]\!](\mathcal{D}\tilde{s}_1)h(\mathcal{D}\tilde{c}_1)\}(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$ (def. $\mathcal{N}$)
$\mathcal{N}[\![p_1\&p_2]\!](\mathcal{D}\tilde{s}_1)h(\mathcal{D}\tilde{c}_1)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$ (def. $\mathcal{D}_C$)
$(\mathcal{D}\langle p_1\&p_2,\tilde{s}_1,h,\tilde{c}_1\rangle)(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$. □

COROLLARY 5.8. For all p, h and $\tilde{s}$, we have

$$\mathcal{O}[\![p]\!]\ h\ \tilde{s} \neq \bot \ \Rightarrow\ \mathcal{D}(\mathcal{O}[\![p]\!]\ h\ \tilde{s}) = \mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s}).$$

PROOF. Immediate from Lemma 5.7. □

The other case to be taken care of is the case that $\mathcal{O}[\![p]\!]\ h\ \tilde{s} = \perp$. We will show in the sequel that this case cannot occur, that is that evaluation of any machine configuration always terminates. It is sufficient to show that there exists a complexity measure $\mathcal{C}$ on machine configurations such that the function *step* decreases this measure for all configurations which are not final.

<u>LEMMA 5.9</u>. If there exists a function $\mathcal{C}$: M $\to$ N such that for all m which are not final we have $\mathcal{C}(m) > \mathcal{C}(step\ m)$; then for all p, h and $\tilde{s}$ we have that $\mathcal{O}[\![p]\!]\ h\ \tilde{s} \neq \perp$.

<u>PROOF</u>. We give two proofs depending on which definition of $\mathcal{P}$ is chosen.

1 (The fixed point definition). It is a well known result that $\mathcal{P} = \bigsqcup_i \phi_i$, where $\phi_0 = \lambda m \cdot \perp$ and $\phi_{i+1} = \lambda m \cdot final(m) \to m,\ \phi_i(step\ m)$. The following property will now be proved by induction on i:

$$(m \neq \perp \wedge \phi_i(m) = \perp) \Rightarrow \mathcal{C}[\![m]\!] \geq i. \qquad \ldots (*)$$

<u>Basis</u>. Trivial.
<u>Induction step</u>. Suppose $\phi_{i+1}(m) = \perp$. This implies that m is not final, so we have $\phi_{i+1}(m) = \phi_i(step\ m)$. The induction hypothesis gives that $\mathcal{C}[\![step\ m]\!] \geq i$ and the property of $\mathcal{C}$ yields that $\mathcal{C}[\![m]\!] > \mathcal{C}[\![step\ m]\!] \geq i$ and therefore $\mathcal{C}[\![m]\!] \geq i+1$.

Having proved (*) we now remark that $(\bigsqcup_i \phi_i)(m) = \perp \Rightarrow \forall i\colon \phi_i(m) = \perp \Rightarrow \forall i\colon \mathcal{C}[\![m]\!] \geq i$ which is clearly impossible.

2 (The row definition). If $\mathcal{P}(m) = \perp$ then there exists an infinite row $m = m_1,\ m_2 = step(m_1), \ldots$ with all m_i not final. That is, there exists an infinite row $m_1, m_2, \ldots$ for which $\mathcal{C}[\![m_i]\!] > \mathcal{C}[\![m_{i+1}]\!]$, and this is not possible because for all m_i we have $\mathcal{C}[\![m_i]\!] \geq 0$. □

The rest of this chapter will be devoted to a definition of a function $\mathcal{C}$ with the desired property. We use the following observations.

1. A machine configuration $m = \langle p,s',h,c,a,q,n,s\rangle$ consists in essence of
   a. A row of pattern components, namely p and the components in the list c.
   b. An alternative a, which is in essence a list, the elements of which are again rows of pattern components.

Combining a. and b., we can view a machine configuration as a *list* $<r_1,...,r_n>$ of *rows of pattern components* ($r_i = <p_{i_1},...,p_{i_k}>$, for some k).

2. Operations, as given by the function *step*, that change the above list are the following:
   a. Operations which change the list into one which is smaller by one element. These are the operations that correspond to a failure in the pattern match. A failure causes backtracking, which amounts to popping a new element from the stack *a*. From this we can conclude that C must be a function that is strictly increasing in the length of the list.
   b. Operations which take one element from the first row of the list. These correspond to cases in which the match succeeds immediately, for instance <u>*nil*</u>, $\bar{n}\$\$v$, $\bar{n}..v$, h' (if the match succeeds). The effect is that the first element is taken from the subsequent *c*.
   We conclude that C must be a strictly increasing function of the length of the first row in the list.
   c. Operations, corresponding to patterns $p\$v$, $p.v$ and $p_1\&p_2$, that add an element to the first row of the list. For these operations the following must hold: $C(<<p_1\&p_2,...>,...>) > C(<<p_1,p_2,...>,...>)$.
   d. Operations, corresponding to $p_1 \vee p_2$ which enlarge the list by one element. The following must hold: $C(<<p_1 \vee p_2>^\wedge rest>,...>) >$ $C(<<p_1>^\wedge rest,<p_2>^\wedge rest,...>)$.

If we now define C(list of rows) $= C(<r_1,...,r_n>) = C(r_1)+...+C(r_n)$ then the property required in a. is satisfied, provided $C(r_i) > 0$. If we take $C(r) = C(<p_1,...,p_k>) = C(p_1)\times...\times C(p_p)$, then also the property from b. holds, provided $C(p) > 1$. Finally, we can meet the restrictions posed in c. and d. by taking $C(p_1\&p_2) = C(p_1)\times C(p_2) + 1$, and $C(p_1 \vee p_2) = C(p_1)+C(p_2) + 1$, respectively.

The above considerations are formalized in the next definition.

<u>DEFINITION 5.10</u> (C).

1. ($C(p)$). $C(h) = C(\underline{nil}) = C(\underline{abort}) = C(\underline{fail}) = C(n\$\$v) = C(n..v) = 2$

   $C(v) = 3$

   $C(p\$v) = C(p.v) = 3\times C(p)$

   $C(*p) = C(p) + 1$

$C(p_1 \vee p_2) = C(p_1)+C(p_2) + 1$

$C(p_1 \& p_2) = C(p_1)\times C(p_2) + 1.$

2. $(C(c))$. $C(\text{<READY>}) = 2$

$C(<p,s_1,h,c>) = C(p)\times C(c).$

3. $(C(a))$. $C(\text{<FAIL>}) = 1$

$C(c\,^\wedge\!<a,q,n>) = C(c)+C(a).$

4. $(C(m))$. $C(m) = 1$ if $final(m)$ holds

$C(c\,^\wedge\!<a,q,n,s>) = C(c)+C(a).$

From this definition the following can be established.

LEMMA 5.11.

1. $\forall p$: $C(p) \geq 2$
2. $\forall c$: $C(c) \geq 2$
3. $\forall a$: $C(a) \geq 1$ and $C(a) = 1 \iff a = \text{<FAIL>}$
4. $C(<p,s_1,h,c,a,q,n,s>) = C(p)\times C(c)+C(a)$
5. $\forall m$: $C(m) \geq 1$ and $C(m) = 1 \iff final(m)$.

PROOF. Easy. □

This lemma can be used to prove the result that we were up to:

LEMMA 5.12. $\neg\, final(m) \Rightarrow C(step\ m) < C(m)$.

PROOF. By cases and easy. The proof has been done informally in the remarks preceding Definition 5.10. □

6. ANOTHER OPERATIONAL SEMANTICS

This chapter shows what the consequences can be for the equivalence proof as given in Chapter 5, if another operational semantics is taken. In this chapter we will give an operational semantics in the style of COOK [2], which has also been used in DE BAKKER [1]. We will use definitions from Chapter 3, but occasionally we will feel free to overwrite the definitions from that chapter, for instance the functions O and C will be defined anew here.

In the new approach we chose to separate again the two phases that can be distinguished in the overall matching process, namely the pattern building phase and the matching phase. For this means we first introduce a new syntactic class, the *pattern structures,* which are the results of pattern building, that is which are patterns without free variables.

$o \in$ *Patstruct*, the *pattern structures.*

This class can be defined by $o ::= h \,|\, \underline{nil} \,|\, \underline{abort} \,|\, \underline{fail} \,|\, o\$v \,|\, o.v \,|\, n\$\$v \,|\, n..v \,|\, *p \,|$
$o_1 \vee o_2 \,|\, o_1 \& o_2.$

Notice the clause $*p$ in this definition. Free variables in p will be bound by the $*$-operator.

We have the following lemma on the denotational meaning of pattern structures which should not be a surprise by now:

LEMMA 6.1. $\forall o \in$ *Patstruct*: $\forall \acute{s}_1, \acute{s}_2$: $N[\![o]\!]\acute{s}_1 = N[\![o]\!]\acute{s}_2$.

PROOF. Easy, by checking the definition of N. □

This lemma justifies the following definition of the meaning function L: *Patstruct* $\to$ *Str* $\to \acute{C} \to \acute{C}$, namely $L[\![o]\!] = N[\![o]\!]\acute{s}$ for some $\acute{s} \in \acute{S}$.

We now introduce the pattern evaluation function E which transforms a pattern expression p, relative to a store $\grave{s}$, into a corresponding pattern structure. This function E: *Pat* $\to \grave{S} \to$ *Patstruct* is defined by cases as:

$$
\begin{aligned}
E[\![p]\!]s &= p \quad \text{for } p \equiv h,\ \underline{nil},\ \underline{abort},\ \underline{fail},\ n\$\$v,\ n..v \text{ and } *p' \\
E[\![v]\!]s &= s(v) \\
E[\![p\$v]\!]s &= (E[\![p]\!]s)\$v \\
E[\![p.v]\!]s &= (E[\![p]\!]s).v \\
E[\![p_1 \& p_2]\!]s &= (E[\![p_1]\!]s)\ \&\ (E[\![p_2]\!]s) \\
E[\![p_1 \vee p_2]\!]s &= (E[\![p_1]\!]s)\ \vee\ (E[\![p_2]\!]s).
\end{aligned}
$$

We have the following lemma, which will be needed in the equivalence proof.

LEMMA 6.2. $\forall p \in$ *Pat* $\forall \grave{s} \in \grave{S}$: $N[\![p]\!](D\grave{s}) = L[\![E[\![p]\!]\grave{s}]\!]$.

PROOF. By cases (induction on the structure of p). The interesting cases are those where $p \notin$ *Patstruct*. We give two examples: $p \equiv v$, $p \equiv p_1 \& p_2$.

1. $\mathcal{N}[\![v]\!](\mathcal{D}\tilde{s}) = \mathcal{N}[\![(\mathcal{D}\tilde{s})(\!|v|\!)]\!](\mathcal{D}\tilde{s})$ = (by Lemma 5.6) $\mathcal{N}[\![\tilde{s}(\!|v|\!)]\!](\mathcal{D}\tilde{s})$ =
   $\mathcal{L}[\![\tilde{s}(\!|v|\!)]\!]$ because $\tilde{s}(\!|v|\!)$ is an element of *Str* and therefore of *Patstruct*. Now, for the same reason, we have by the definition of $\mathcal{E}$ that $\mathcal{E}[\![\tilde{s}(\!|v|\!)]\!] = \tilde{s}(\!|v|\!)$, and we are ready.
2. $\mathcal{N}[\![p_1\&p_2]\!](\mathcal{D}\tilde{s})\ h\ \tilde{c} = \mathcal{N}[\![p_1]\!](\mathcal{D}\tilde{s})\ h\ \{\mathcal{N}[\![p_2]\!](\mathcal{D}\tilde{s})\ h\ \tilde{c}\}$ = (ind.)
   $= \mathcal{L}[\![\mathcal{E}[\![p_1]\!]\tilde{s}]\!]\ h\ \{\mathcal{L}[\![\mathcal{E}[\![p_2]\!]\tilde{s}]\!]\ h\ \tilde{c}\}$ = (def. $\mathcal{L}$)
   $= \mathcal{N}[\![\mathcal{E}[\![p_1]\!]\tilde{s}]\!]\ \acute{s}_1\ h\ \{\mathcal{N}[\![\mathcal{E}[\![p_2]\!]\tilde{s}]\!]\ \acute{s}_1\ h\ \tilde{c}\}$ = (def. $\mathcal{N}$)
   $= \mathcal{N}[\![(\mathcal{E}[\![p_1]\!]\ \tilde{s})\ \&\ (\mathcal{E}[\![p_2]\!]\tilde{s})]\!]\ \acute{s}_1\ h\ \tilde{c}$ = (def. $\mathcal{L}$ and $\mathcal{E}$)
   $\mathcal{L}[\![\mathcal{E}[\![p_1\&p_2]\!]\tilde{s}]\!]\ h\ \tilde{c}$. □

We will next give the new operational semantics. The operational meaning function $\mathcal{O}$ (which has again functionality $\mathcal{O}$: *Pat* → *Str* → $\tilde{S}$ → $\grave{R}$) is defined in terms of an auxiliary function $\mathcal{P}$. The function $\mathcal{P}$ takes (among others) a pattern structure and delivers a finite row of intermediate results which are triples. Each triple consists of a cursor position (a numeral), a store and a queue of conditional assignments accumulated. Such a row of intermediate results can be seen as the trace left by the pattern matching process. We thus need the following definition:

$i \in I = (\mathit{Num} \cup \{\text{FAIL},\text{ABORT}\}) \times \tilde{S} \times \grave{L}$, the class of *intermediate results*.

We furthermore define the tail function κ which takes the last element of a list: $\kappa\langle e_1,\ldots,e_n\rangle = e_n$.

We now define the function $\mathcal{P}$: *Patstruct* → *Str* → I → I^+ inductively as follows:

1. $\mathcal{P}[\![h']\!]\ h\ \langle n,s,q\rangle = \begin{cases}\langle \mathcal{D}^{-1}\ n',s,q\rangle & \text{if } h' \equiv h[\mathcal{D}n{:}n'] \\ \langle \text{FAIL},s,q\rangle & \text{otherwise}\end{cases}$
2. $\mathcal{P}[\![\underline{nil}]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle$
3. $\mathcal{P}[\![\underline{fail}]\!]\ h\ \langle n,s,q\rangle = \langle \text{FAIL},s,q\rangle$
4. $\mathcal{P}[\![abort]\!]\ h\ \langle n,s,q\rangle = \langle \text{ABORT},s,q\rangle$
5. $\mathcal{P}[\![o\$v]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge\mathcal{P}[\![o\&(n\$\$v)]\!]\ h\ \langle n,s,q\rangle$
6. $\mathcal{P}[\![o.v]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge\mathcal{P}[\![o\&(n..v)]\!]\ h\ \langle n,s,q\rangle$
7. $\mathcal{P}[\![n'\$\$v]\!]\ h\ \langle n,s,q\rangle = \langle n,s^\wedge\langle v,h[\mathcal{D}n'{:}\mathcal{D}n]\rangle,q\rangle$
8. $\mathcal{P}[\![n'..v]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q^\wedge\langle v,h[\mathcal{D}n'{:}\mathcal{D}n]\rangle\rangle$

9. $\mathcal{P}\llbracket o_1 \vee o_2 \rrbracket\ h\ \langle n,s,q\rangle =$

   $n' \neq \mathrm{FAIL} \rightarrow \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_1 \rrbracket\ h\ \langle n,s,q\rangle, \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_1 \rrbracket\ h\ \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_2 \rrbracket\ h\ \langle n,s',q\rangle,$

   where $\langle n',s',q\rangle = \kappa(\mathcal{P}\llbracket o_1 \rrbracket\ h\ \langle n,s,q\rangle)$

10. $\mathcal{P}\llbracket *p \rrbracket\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket \mathcal{E}\llbracket p \rrbracket s \rrbracket\ h\ \langle n,s,q\rangle$

11. $\mathcal{P}\llbracket \bar{o} \& o_2 \rrbracket\ h\ \langle n,s,q\rangle =$ (where $\bar{o} \equiv h'$, $\underline{nil}$, $\underline{abort}$, $\underline{fail}$, $n''\$\$v$ or $n''..v$)

   $n' = \mathrm{FAIL}, \mathrm{ABORT} \rightarrow \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket \bar{o} \rrbracket\ h\ \langle n,s,q\rangle,$

   $\langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket \bar{o} \rrbracket\ h\ \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_2 \rrbracket\ h\ \langle n',s',q'\rangle,$

   where $\langle n',s',q'\rangle = \kappa(\mathcal{P}\llbracket \bar{o} \rrbracket\ h\ \langle n,s,q\rangle)$

12. $\mathcal{P}\llbracket (o_1 \& o_2) \& o_3 \rrbracket\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_1 \& (o_2 \& o_3)) \rrbracket\ h\ \langle n,s,q\rangle$

13. $\mathcal{P}\llbracket (o_1 \vee o_2) \& o_3 \rrbracket\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket (o_1 \& o_3) \vee (o_2 \& o_3) \rrbracket\ h\ \langle n,s,q\rangle$

14. $\mathcal{P}\llbracket (o_1 \$ v) \& o_2 \rrbracket\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_1 \& ((n\$\$v) \& o_2) \rrbracket\ h\ \langle n,s,q\rangle$

15. $\mathcal{P}\llbracket (o_1 . v) \& o_2 \rrbracket\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket o_1 \& ((n..v) \& o_2) \rrbracket\ h\ \langle n,s,q\rangle$

16. $\mathcal{P}\llbracket (*p) \& o_2 \rrbracket\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle {}^\wedge \mathcal{P}\llbracket (\mathcal{E}\llbracket p \rrbracket s) \& o_2 \rrbracket\ h\ \langle n,s,q\rangle.$

Remarks.

The essential difference with the definition of *step* in Chapter 3 is that here we do not use explicit stacks (c and a). The alternatives remaining are remembered implicitly as can be seen from clause 9: matching against $o_1 \vee o_2$ amounts to matching against o_1 if this match succeeds or is aborted. Otherwise it is the same as matching against o_1 and afterwards against o_2 starting with the correct intermediate result $\langle n,s',q\rangle$.

The subsequents to be applied later are in principle retained in the pattern component itself. Clauses 11 through 16 all deal with patterns of the form $o \& o'$. Clause 11 ($o \equiv \bar{o}$) gives the case where o does not have implicit alternatives which means that no backtracking to o is possible. In that case matching against o is tried, and we go on if this match succeeds. In all other cases (12 - 16) we have to find out which elementary pattern component has to be matched against first. We solved this by first decomposing the first operand of $o \& o'$ until an elementary pattern is reached. For instance the pattern structure $(((h_1 \vee h_2)\$v) \& \underline{fail}) \& o'$ will be rewritten as follows (where we assume that matching starts with cursor position given by the numeral n):

$(((h_1 \vee h_2)\$v) \& \underline{fail}) \& o' \rightarrow$	(clause 12)
$((h_1 \vee h_2)\$v) \& (\underline{fail} \& o') \rightarrow$	(clause 14)
$(h_1 \vee h_2) \& ((n\$\$v) \& (\underline{fail} \& o')) \rightarrow$	(clause 13)
$(h_1 \& ((n\$\$v) \& (\underline{fail} \& o'))) \vee (h_2 \& ((n\$\$v) \& (\underline{fail} \& o')).$	

Now we use clause 9, and first investigate the first operand of this disjunction which is $(h_1 \,\&\, ((n\$\$v) \,\&\, (\underline{fail} \,\&\, o')))$. On this pattern structure we then apply clause 11, clause 1, etc.

Notice also clause 10 and 16 of the definition. If, while matching, the scanner encounters an $*$-operator, first the corresponding pattern component is evaluated using $\mathcal{E}$, before proceeding.

The claim on the functionality of $\mathcal{P}$, in particular that $\mathcal{P}$ yields values in I^+, and also the statement that the above definition is an inductive one, has to be justified. We do this by presenting a complexity measure $\mathcal{C}$ on pattern structures such that all structures occurring in the right-hand sides of the clauses of the definition of $\mathcal{P}$ have smaller $\mathcal{C}$-values than the o's in the corresponding left-hand sides. The following function $\mathcal{C}$, defined on $\mathcal{P}at$ by structural induction, does the job:

$$\begin{aligned}
&\mathcal{C}[\![h]\!] = \mathcal{C}[\![v]\!] = \mathcal{C}[\![\underline{nil}]\!] = \mathcal{C}[\![\underline{abort}]\!] = \mathcal{C}[\![\underline{fail}]\!] = \mathcal{C}[\![n\$\$v]\!] = \mathcal{C}[\![n..v]\!] = 1\\
&\mathcal{C}[\![p\$v]\!] = \mathcal{C}[\![p.v]\!] = 2\mathcal{C}[\![p]\!] + 2\\
&\mathcal{C}[\![*p]\!] = \mathcal{C}[\![p]\!] + 1\\
&\mathcal{C}[\![p_1\&p_2]\!] = 2\mathcal{C}[\![p_1]\!] + \mathcal{C}[\![p_2]\!]\\
&\mathcal{C}[\![p_1 \vee p_2]\!] = \max\{\mathcal{C}[\![p_1]\!], \mathcal{C}[\![p_2]\!]\} + 1.
\end{aligned}$$

We are now ready to define the function $\mathcal{O}$ with functionality $\mathcal{O}\colon \mathcal{P}at \to \mathcal{S}tr \to S \to R$ as follows:

$$\begin{aligned}
\mathcal{O}[\![p]\!]\; h\; s = (n' = \text{ABORT},\text{FAIL}) \to \langle n',s'\rangle, \langle n', s'^{\wedge}q'\rangle,\\
\text{where } \langle n',s',q'\rangle = \kappa(\mathcal{P}[\![\mathcal{E}[\![p]\!]s]\!]\; h\; \langle 0,s,\langle\rangle\rangle).
\end{aligned}$$

In order to be able to prove an equivalence result similar to the one in Chapter 5, we need some auxiliary facts:

<u>LEMMA 6.3.</u>

1. $\mathcal{N}[\![(p_1\&p_2)\,\&\,p_3]\!] = \mathcal{N}[\![p_1\,\&\,(p_2\&p_3)]\!]$
2. $\mathcal{N}[\![(p_1 \vee p_2)\,\&\,p_3]\!] = \mathcal{N}[\![(p_1\&p_3) \vee (p_2\&p_3)]\!]$
3. $\mathcal{N}[\![(p_1\$v)\,\&\,p_2]\!]\; s_1\; h\; c\; a\; q\; n\; s = \mathcal{N}[\![p_1\,\&\,((\mathcal{D}^{-1}n\$\$v)\,\&\,p_2)]\!]\; s_1\; h\; c\; a\; q\; n\; s$
4. $\mathcal{N}[\![(p_1.v)\,\&\,p_2]\!]\; s_1\; h\; c\; a\; q\; n\; s = \mathcal{N}[\![p_1\,\&\,((\mathcal{D}^{-1}n..v)\,\&\,p_2)]\!]\; s_1\; h\; c\; a\; q\; n\; s$
5. $\mathcal{N}[\![(*p_1)\,\&\,p_2]\!]\; s_1\; h\; c\; a\; q\; n(\mathcal{D}\bar{s}) = \mathcal{N}[\![(\mathcal{E}[\![p_1]\!]\bar{s})\,\&\,p_2]\!]\; s_1\; h\; c\; a\; q\; n\; (\mathcal{D}\bar{s})$.

PROOF.

1 and 2. By writing out the respective clauses in the definition of $\mathcal{N}$.

3 and 4. By Lemma 4.1, by result 1 of this lemma, and by the fact that

$\mathcal{N}[\![p_1]\!] = \mathcal{N}[\![p_2]\!]$ implies $\mathcal{N}[\![p_1\&p_3]\!] = \mathcal{N}[\![p_2\&p_3]\!]$.

5. $\mathcal{N}[\![(*p_1)\ \&\ p_2]\!]\ s_1\ h\ c\ a\ q\ n\ (\mathcal{D}\tilde{s})$ = (def. $\mathcal{N}$)

$\mathcal{N}[\![*p_1]\!]\ s_1\ h\ \{\mathcal{N}[\![p_2]\!]\ s_1\ h\ c\}\ a\ q\ n\ (\mathcal{D}\tilde{s})$ = (def. $\mathcal{N}$)

$\mathcal{N}[\![p_1]\!](\mathcal{D}\tilde{s})\ h\ \{\mathcal{N}[\![p_2]\!]\ s_1\ h\ c\}\ a\ q\ n\ (\mathcal{D}\tilde{s})$ = (Lemma 6.2, 6.1)

$\mathcal{N}[\![\mathcal{E}[\![p_1]\!]\tilde{s}]\!]\ s_1\ h\ \{\mathcal{N}[\![p_2]\!]\ s_1\ h\ c\}\ a\ q\ n\ (\mathcal{D}\tilde{s})$ = (def. $\mathcal{N}$)

$\mathcal{N}[\![(\mathcal{E}[\![p_1]\!]\tilde{s})\ \&\ p_2]\!]\ s_1\ h\ c\ a\ q\ n\ (\mathcal{D}\tilde{s})$. □

Now if we want to prove $\mathcal{O}$ and $\mathcal{M}$ equivalent it appears that we have to formulate a rather complicated induction hypothesis relating $\mathcal{N}$ and $\mathcal{P}$. This is due to the fact that in the operational definition of this chapter no counterparts of the entities c and a from the denotational definition exist. We have to capture the effects that subsequents and alternatives may have by formulating the following induction hypothesis. The main trick is that we capture the effect of the alternative a in the denotational definition by quantifying over all alternatives.

LEMMA 6.4. For all o, h, $\tilde{n}$, $\tilde{s}$ and $\tilde{q}$ we have

$$\mathcal{P}[\![o]\!]\ h\ \langle\tilde{n},\tilde{s},\tilde{q}\rangle \cong \lambda a\cdot \mathcal{L}[\![o]\!]\ h\ \mathit{ready}\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$$

where *list* $\cong \phi$ iff

$$\left[\begin{array}{lll} \underline{\text{either}} & \kappa(\mathit{list}) = \langle\text{FAIL},\tilde{s}_1,\tilde{q}_1\rangle & \text{and } \phi = \lambda a\cdot a(\mathcal{D}\tilde{s}_1) \\ \underline{\text{or}} & \kappa(\mathit{list}) = \langle\text{ABORT},\tilde{s}_1,\tilde{q}_1\rangle & \text{and } \phi = \lambda a\cdot\langle\text{ABORT},(\mathcal{D}\tilde{s}_1)\rangle \\ \text{or} & \kappa(\mathit{list}) = \langle\tilde{n}_1,\tilde{s}_1,\tilde{q}_1\rangle & \text{and } \phi = \lambda a\cdot\langle\mathcal{D}\tilde{n}_1,(\mathcal{D}\tilde{q}_1)(\mathcal{D}\tilde{s}_1)\rangle \end{array}\right].$$

PROOF. By induction on the $\mathcal{C}$-complexity of o. We have to distinguish all cases as occurring in the definition of $\mathcal{P}$ which is tedious. So we give a few typical examples. We define $\mathit{lhs} = \kappa(\mathcal{P}[\![o]\!]\ h\ \langle\tilde{n},\tilde{s},\tilde{q}\rangle)$ and $\mathit{rhs} = \lambda a\cdot \mathcal{L}[\![o]\!]\ h\ \mathit{ready}\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

2. (<u>*nil*</u>) $\mathit{lhs} = \langle\tilde{n},\tilde{s},\tilde{q}\rangle$ and $\mathit{rhs} = \lambda a\cdot\langle\mathcal{D}\tilde{n},(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})\rangle$

3. (<u>*fail*</u>) $\mathit{lhs} = \langle\text{FAIL},\tilde{s},\tilde{q}\rangle$ and $\mathit{rhs} = \lambda a\cdot a(\mathcal{D}\tilde{s})$

8. $(n'..v)$ We have $lhs = \langle\tilde{n},\tilde{s},\tilde{q}\,{}^\wedge\langle v,h[\mathcal{D}n':\mathcal{D}\tilde{n}]\rangle\rangle$ and

$$rhs = \lambda a\bullet \mathcal{L}[\![n'..v]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$$
$$\lambda a\bullet ready\ a\ (\lambda\tilde{s}\bullet((\mathcal{D}\tilde{q})\tilde{s})\{h[\mathcal{D}n':\mathcal{D}\tilde{n}]/v\})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$$
$$\langle\mathcal{D}\tilde{n},((\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s}))\{h[\mathcal{D}n':\mathcal{D}\tilde{n}]/v\}\rangle.$$

So we have to prove that this is equal to $(\mathcal{D}(\tilde{q}\,{}^\wedge\langle v,h[\mathcal{D}n':\mathcal{D}\tilde{n}]\rangle))(\mathcal{D}\tilde{s})$ and this is true by the definition of $\mathcal{D}_Q$.

9. $(o_1\vee o_2)$. Let $\langle\tilde{n}_2,\tilde{s}_2,\tilde{q}_2\rangle = \kappa(\mathcal{P}[\![o_1]\!]\ h\ \langle\tilde{n},\tilde{s},\tilde{q}\rangle)$.

A. n_2 = ABORT.

Then $lhs = \langle\text{ABORT},\tilde{s}_2,\tilde{q}_2\rangle$ and by induction we have for all a':

$\mathcal{L}[\![o_1]\!]\ h\ ready\, a'\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) = \langle\text{ABORT},\mathcal{D}\tilde{s}_2\rangle$.

This holds in particular for $a' = \mathcal{L}[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})$ and we thus get for all a: $\mathcal{L}[\![o_1\vee o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) = \langle\text{ABORT},\mathcal{D}\tilde{s}_2\rangle$.

B. $n_2 \in \mathit{Num}$.

The argument is similar to that in case A.

C. n_2 = FAIL.

We have $lhs = \kappa(\mathcal{P}[\![o_2]\!]\ h\ \langle\tilde{n},\tilde{s}_2,q\rangle)$.

By induction we have for all a' that

$\mathcal{L}[\![o_1]\!]\ h\ ready\ a'(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) = a'(\mathcal{D}\tilde{s}_2)$. This holds in particular for $a' = \mathcal{L}[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})$ and we get

$$rhs = \lambda a\bullet \mathcal{L}[\![o_1\vee o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$$
$$\lambda a\bullet \mathcal{L}[\![o_1]\!]\ h\ ready\ \{\mathcal{L}[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})\}(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$$
$$\lambda a\bullet \mathcal{L}[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}_2).$$

Now we can apply the induction hypothesis, for $\mathcal{C}[\![o_2]\!] < \mathcal{C}[\![o_1\vee o_2]\!]$.

12. $((o_1\&o_2)\ \&\ o_3)$. Use Lemma 6.3.1, and the induction hypothesis (notice that $\mathcal{C}[\![o_1\ \&\ (o_2\&o_3)]\!] < \mathcal{C}[\![(o_1\&o_2)\ \&\ o_3]\!]$.

16. $((*p)\ \&\ o_2)$. Use Lemma 6.3.5 and the induction hypothesis. □

<u>THEOREM 6.5</u>. For all p, h and $\tilde{s}$ we have $\mathcal{D}(\mathcal{O}[\![p]\!]\ h\ \tilde{s}) = \mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s})$.

<u>PROOF</u>. We have $\mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = \mathcal{N}[\![p]\!](\mathcal{D}\tilde{s})\ h\ ready\ fail\ (\lambda s\bullet s)\ 0\ (\mathcal{D}\tilde{s}) =$
$= \mathcal{L}[\![\mathcal{E}[\![p]\!]\tilde{s}]\!]\ h\ ready\ fail\ (\lambda s\bullet s)\ 0\ (\mathcal{D}\tilde{s})$ by Lemma 6.2.

Let $\kappa(\mathcal{P}[\![\mathcal{E}[\![p]\!]\tilde{s}]\!]\ h\ \langle 0,\tilde{s},\langle\rangle\rangle = \langle\tilde{n}_1,\tilde{s}_1,\tilde{q}_1\rangle$. There are three cases.

1. n_1 = FAIL. By Lemma 6.4.: $\mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = fail\ (\mathcal{D}\tilde{s}_1) = \langle\text{FAIL},\mathcal{D}\tilde{s}_1\rangle$.

By the definition of $\mathcal{O}$ we have $\mathcal{O}[\![p]\!]\ h\ \tilde{s} = \langle\text{FAIL},\tilde{s}_1\rangle$.

2. If n_1 = ABORT, then by Lemma 6.4 we have that $\mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = \langle\text{ABORT},\mathcal{D}\tilde{s}_1\rangle$ and by the definition of $\mathcal{O}$ we have $\mathcal{O}[\![p]\!]\ h\ \tilde{s} = \langle\text{ABORT},\tilde{s}_1\rangle$.

3. If $n_1 \in \mathit{Num}$ then Lemma 6.4 gives $\mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = \langle\mathcal{D}\tilde{n}_1,(\mathcal{D}\tilde{q}_1)(\mathcal{D}\tilde{s}_1)\rangle$, while the definition of $\mathcal{O}$ yields $\mathcal{O}[\![p]\!]\ h\ \tilde{s} = \langle\tilde{n}_1,\tilde{s}_1{}^\wedge\tilde{q}_1\rangle$. Now by Lemma 5.2: $\mathcal{D}\langle\tilde{n}_1,\tilde{s}_1{}^\wedge\tilde{q}_1\rangle = \langle\mathcal{D}\tilde{n}_1,(\mathcal{D}\tilde{q}_1)(\mathcal{D}\tilde{s}_1)\rangle$ and we are ready. □

7. CONCLUDING REMARKS AND ACKNOWLEDGEMENTS

This report presents the first results of a project in which we aim to study various semantical aspects of the matching process in SNOBOL4. The next step to be taken is to allow patterns as values of variables, instead of strings as was the case here. This will lead to a (denotational) store S which will be a function from variables to patterns, where patterns are modelled by functions which describe (amongst others) store transformations. This suggests a reflexive (circular) definition of the domain of stores, and an equivalence proof like the one given here will be much harder to construct.

This is why we chose to do some "ground work" first, and this paper presents the results of it. We chose the SNOBOL subset such that all essential aspects of pattern matching are reflected in it, apart from the idea that patterns can be values of variables.

The denotational semantics given here should be compared with the one given by TENNENT [9] which is far more complicated due to the fact that a much larger subset of SNOBOL4 is involved here. Our semantics can be viewed as a simplification of Tennent's, resulting in a semantics that describe the matching process clearly with no more tools and complications than needed.

In Chapter 6 we showed that one has to be careful in designing an operational semantics, if one wishes to prove an equivalence result. The semantics of Chapter 3 is inspired by the operational semantics in STOY [8]. We borrowed his idea to carefully provide for each denotational notion a corresponding operational notion. For instance our operational semantics uses a class of subsequents and a class of alternatives which correspond to the denotational domains C and A. This made the equivalence proof manageable, as can be seen when one compares the proof in Chapter 5 with the one in Chapter 6 where an operational semantics was used which was less carefully designed.

Apart from Stoy's, the papers by GIMPEL [3] and PAGAN [6] should be mentioned. They provided many useful details about the peculiarities of the SNOBOL language.

Finally, I like to mention Jaco de Bakker, who has read an earlier version of this paper and who came up with useful comments, and also Ruurd Kuiper with whom I had fruitful discussions on the topics treated here.

REFERENCES

[1] BAKKER, J.W. DE, *Mathematical theory of program correctness,* Prentice Hall Int. (1980).

[2] COOK, S.A., *Soundness and completeness of an axiom system for program verification,* SIAM J. on Computing, Vol. 7, nr. 1, pp. 70-90 (1978).

[3] GIMPEL, J.F., *A theory of discrete patterns and their implementation in SNOBOL4,* Comm. of the ACM, Vol. 16, nr. 2, pp. 91-100 (1973).

[4] LANDIN, P.J., *The mechanical evaluation of expressions,* Computer J., Vol. 6, nr. 4, pp. 308-320 (1964).

[5] MILNE, R. & C. STRACHEY, *A theory of programming language semantics,* Chapman and Hall, London and Wiley, New York, 2 vols. (1976).

[6] PAGAN, F.G., *Formal semantics of a SNOBOL4 subset,* Computer Languages, Vol. 3, pp. 13-30 (1978).

[7] STOY, J.E., *Denotational semantics - the Scott-Strachey approach to programming language theory,* M.I.T. Press, Cambridge, Mass. (1977).

[8] STOY, J.E., *The congruence of two programming language definitions,* to appear.

[9] TENNENT, R.D., *Mathematical semantics and design of programming languages,* University of Toronto, Technical Report nr. 59 (1973).