Pattern Recognition of Noisy Sequences of Behavioural Events Using Functional Combinators

To appear in the Computer Journal 37(5)

A. N. Clark, Computer Systems Division, GEC-Marconi Research,
Great Baddow, Chelmsford, Essex, CM2 8HN, Great Britain
e-mail: anc@gec-mrc.co.uk, tel.: (0245) 473331

1 Abstract

Vision systems are now delivering real-time tracked and classified data from which behaviours can be inferred. Behavioural specifications of observable entities are viewed as attribute grammars with general constraints on the values of the attributes. The characteristic features of realistic input data to a behavioural recogniser are missing, inserted and noisy values. Behaviour recognition is defined in terms of a simple language which is used to express behaviour specifications. The meaning of a specification is given by a mapping to the set of observable value sequences which exhibit the specified behaviour. A computational mechanism for behaviour recognition, which is consistent with the meaning of behaviour specifications, is given using a simple functional programming language which has been enriched with non-deterministic features. The non-determinism is controlled using a simple belief mechanism.

2 Introduction

Conceptual processing of observed data [17] involves constructing high level structured symbolic descriptions in terms of low level, unstructured and incomplete input. Observed data will be a stream of simple discrete events for single entities, *e.g.* object o_1 enters the scene at time t_1 and is classified as a *truck*. A behaviour for a single entity can be described as a sequence of observed low level events, *e.g.* if a truck on the ground at an airport enters the scene then arrives at the aft-hold of an aircraft, waits for 10 minutes[1] and then leaves the scene, then the truck is said to have exhibited a behaviour consistent with loading baggage onto the aircraft.

Behaviours are readily described using scripts [19] which represent perfect sequences of events. Given perfect input data, ordered with respect to time, the activity of behaviour recognition can be viewed as parsing, with the scripts taking the place of the grammar and the observed events in place of the input tokens. This analogy is extended by noting that the input data will be structured, *e.g.* an input token may contain a time, a simple action, an entity identifier and a classification. Behaviour recognition of structured input data is viewed as parsing with respect to an attribute grammar [14] [8] with general constraints on the values of the attributes.

Unfortunately, the processes by which real world events are transformed into input data are far from perfect. For example classifications of entities will be inconsistent, particularly as the entity is initially tracked, not all events will be detected and incorrect events will be hallucinated. The characteristic features of real world input data will be

- missing data - some events which are expected will not be detected and must be hypothesised.
- junk data - spurious events which are unexpected will be received.
- noisy data - values will not always be exact or consistent.

This paper describes work which has been carried out as part of the ESPRIT VIEWS project [3] [5] [4] which has developed a real-time single camera vision system which classifies and tracks vehicles in a scene. The behaviour of vehicles servicing an aircraft have been specified using an attribute grammar. The combined system demonstrates real-time behaviour recognition. The paper is structured as follows:

- §3 describes related work which is compared to the described approach in §9.
- A simple language for specifying behaviours is defined in §4 and given a semantics in terms of a mapping from each specification to the set of observable value sequences which will be said to "exhibit the behaviour". The mapping will determine meanings for the characteristic features of noise, junk and hypotheses.
- Although the language gives a precise meaning for the term "behaviour specification" it does not help in recognising the behaviours given sequences of observed values. Recognition will be performed using a parser which is implemented in a simple function language which is defined in §5.

[1] The interval of 10 minutes is completely arbitrary and could be a parameter of the behaviour if desired.

- §6 defines the parsing machinery using the functional language which will perform behaviour recognition.
- §7 gives a simple example of a behaviour specification and value sequences which exhibit the characteristic features of real world data.
- The parsers which are built using the techniques in §6 will not, in general, be complete with respect to the meaning of the language defined in §4. §8 shows three different mechanisms for controlling the parse.

3 Related Work

[9] is a comprehensive introduction to the field of pattern recognition using syntactic methods and covers techniques for string and stochastic parsing in particular. [20] describes attribute grammars applied to pattern recognition where the terminal symbols are associated with attribute vectors, non-terminals construct attribute vectors as functions of their components and complete alternative parses are ruled out on the basis of their overall attribute vector. [1] describes the different types of error which can be observed during pattern recognition. [7] observes that a statistical measure can be associated with the different types of error and alternative parses can be distinguished on the basis of a confidence measure. [6] describes pattern recognition using graph grammars which allow sharing in between terminal symbols. [22] describes a method of parsing incomplete input data by growing islands and coalescing islands which are sufficiently close together in the parse.

Parsing using functional combinators has been described in [2] and [11] and are generalised to include attributes in [12] and [8]. [18] and [21] show how monads can be used to implement parsers and hide error information.

4 Behaviours as grammars

A grammar represents a formal language which is the set of strings which the grammar generates. A behaviour specification represents a behaviour which is the set of observable event sequences which is generated from it. Parsing with respect to a grammar is viewed as testing a given string for set membership in the language generated from the grammar. Of course it is too costly to generate the entire set and then test for membership, so parsers test each element of the input string in turn, generating only a small portion of the language as they go.

This section will show how behaviour specifications are viewed as grammars which generate a formal language and how behaviour recognition is viewed as set membership. §6 will show how conventional parsing techniques can be modified to generate a behaviour recogniser which conforms to set membership but generates the set lazily.

Figure 1 shows the grammar for a language which specifies behaviours. The language will be described informally and then given a semantics by defining a mapping which will translate any rule $r \in R$ to a set of value sequences which the rule specifies. The syntactic categories are: rules R, patterns P, rule bodies B, expressions E, identifiers I, numbers N and strings S. A pattern P may be a data constructor $k \in K$ followed by a pattern, an identifier, a number, a string, an n-tuple of patterns or a classification, which is an @ character followed by a classification name string.

A rule $[\![p : - b]\!] \in R$ is a behaviour specification which will be a set of observable value sequences. Each of the observable sequences will be associated with a value which is defined by the pattern p. For example if the rule is $[\![(x, y) : - b]\!]$ for identifiers x and y and a body b then b will define sequences of values in terms of the identifiers x and y (and possibly other identifiers too), the rule will represent each value sequence tagged with a pair containing the value of x and value of y which was used to generate that particular sequence.

A body is constructed using ;, |, \$, ? and &. In the following description, the *current* rule will be the innermost rule which textually contains the body construct in question. The $[\![b_1; b_2]\!] \in B$ construct will specify that the sequences of values denoted by b_1 must be observed before the sequences denoted by b_2. The $[\![b_1|b_2]\!] \in B$ construct will specify that either the sequences denoted by b_1 or those denoted by b_2 must be observed. The $[\![\$p]\!] \in B$ construct will specify that the next value to be observed will be described by the pattern p. The $[\![\&rp]\!] \in B$ construct will specify that the next collection of sequences will be specified by the rule r which denotes a collection of sequences each of which is tagged with a value. Each value which is associated with an r sequence will be *unified* with p to produce the values of the identifiers (viewed as variables) associated with the sequences in the context of the current rule. The $[\![?ie]\!] \in B$ construct specifies a constraint on the values of the variables for each sequence in the current rule. i will be the name of a predicate and e will be an expression in terms of the identifiers of the current rule. The sequences which are specified by the current rule must have identifier values which satisfy the predicate. e is an expression which is an identifier, a string or an operator applied to operand expressions *e.g.* $[\![e_1 + e_2]\!]$.

$$
\begin{aligned}
R &::= \quad P : -B \\
P &::= \quad KP \mid I \mid N \mid S \mid (P_1, P_2, \ldots, P_n) \mid @S \\
B &::= \quad B_1 ; B_2 \mid B_1 | B_2 \mid \$P \mid ?IE \mid \&RP \\
E &::= \quad E_1 + E_2 \mid \ldots \mid I \mid S \mid N \mid @S
\end{aligned}
$$

Figure 1: A language for specifying behaviour grammars

$$D[\![p : -b]\!] = \{(\omega, v, c) \mid (\omega, e, c) \in D(b), v = e(p)\}$$

$$D[\![b_1 ; b_2]\!] = \{(\omega_1\omega_2, e_1 \oplus e_2, c_1 + c_2) \mid (\omega_1, e_1, c_1) \in D(b_1), (\omega_2, e_2, c_2) \in D(b_2), \mathit{agree}(e_1, e_2)\}$$

$$D[\![b_1 | b_2]\!] = D(b_1) \cup D(b_2)$$

$$D[\![?ie]\!] = \{(\gamma, e, c) \mid (v, e, c) \in D(e), i(v)\}$$

$$D[\![e_1 + e_2]\!] = \{(v_1 + v_2, e_1 \oplus e_2, c_1 + c_2) \mid (v_1, e_1, c_1) \in D(e_1), (v_2, e_2, c_2) \in D(e_2)\}$$

$$D[\![i]\!] = \{(v, i \mapsto v, 0) \mid v \in V\}$$

$$D[\![s]\!] = \{(s, \{\}, 0)\}$$

$$D[\![@s]\!] = \{(@s', \{\}, c) \mid s'' \in T, s' = \mathit{glb}(s, s''), s' \neq \perp, c = \mathit{diff}(s, s')\}$$

$$D[\![\$p]\!] = J(D(p) \cup \{(\gamma, e, c+1) \mid (\omega, e, c) \in D(p)\})$$

$$D[\![kp]\!] = \{(kv, e, c) \mid (v, e, c) \in D(p)\}$$

$$
\begin{aligned}
D[\![(p_1, \ldots, p_n)]\!] = \{&((v_1, \ldots, v_n), e_1 \oplus \ldots \oplus e_n, c_1 + \ldots + c_n) \mid \\
&(v_1, e_1, c_1) \in D(p_1), \ldots, (v_n, e_n, c_n) \in D(p_n), \\
&\forall e, e' \in \{e_1, \ldots, e_n\} \bullet \mathit{agree}(e, e')\}
\end{aligned}
$$

$$D[\![\&rp]\!] = \{(\omega, e, c) \mid (\omega, v, c) \in D(r), e(p) = v\}$$

$$J(s) = s \cup J(s')$$

where

$$
\begin{aligned}
s' = &\{(v\omega, e, c+1) \mid (\omega, e, c) \in s, v \in V\} \cup \\
&\{(\omega v, e, c+1) \mid (\omega, e, c) \in s, v \in V\}
\end{aligned}
$$

Figure 2: The semantics of behaviour specifications

A rule is given a precise meaning by translating it to the set of value sequences which it specifies using the operator D which is defined in figure 2. The definition of D uses set comprehension notation:

$$\{v \mid p_1, p_2, \ldots, p_n\}$$

where v is a value expression and p_i are predicate expressions. A set comprehension will contain some identifiers and will denote the set which is constructed by substituting all possible combinations of values for the identifiers in v such that the corresponding predicate expressions are true. D also uses a universally quantified boolean expression

$$\forall i \in S \bullet e$$

The meaning of a rule will be a triple

$$(\omega, v, c)$$

where ω is a value sequence, v is the value specified by the pattern in the rule and c is an integer which represents how close the sequence ω is to that which was specified in the rule using the $ constructs. A c value will be referred to as a measure of "closeness", the lower the value – the closer the match. The meaning of a body construct will be a triple

$$(\omega, e, c)$$

where ω and c are a value sequence and closeness measure respectively and e is an *environment* containing associations between the identifiers in the current rule and the values which they have been given to produce the value sequence ω. An environment will be either empty, $\{\}$, a single binding from an identifier to a value $i \mapsto v$ or a concatenation of environments $e_1 \oplus e_2$. The result of substituting values for all identifiers bound in an environment e which occur in a pattern p is represented as $e(p)$. The predicate *agree* is true of two environments when the identifiers which both environments have in common are bound to the same values in both environments. The operator *diff* will produce a numeric value which represents the degree of difference between two classification names. V is the set of all values which can occur in value sequences and T is the set of all classification type names.

Value sequences will correspond to the actions which are performed by observed entities and will contain classifications for those entities. In a realistic situation, the software which performs the classification may make mistakes and classify an entity of type t as a different, but related, type t'. The classification types are arranged in a lattice structure with a top and bottom element. Given two classification types, there will be a greatest lower bound $glb(t, t')$ and a closeness measure $diff(t, t')$ The denotation of a classification in figure 2 is defined to be all of the related classifications and a closeness measure.

Given a behaviour specification for a single observable value $[\![\$p]\!]$, the pattern p will determine a set of possible values which fit the specification. If the values which are presented to the behaviour recognition system could be guaranteed to be perfect then the set of single values corresponding to all the ways in which the identifiers in p can be bound would be the meaning of $[\![\$p]\!]$. Unfortunately, in a realistic situation, the software which provides the observed values may get things wrong and the observed entity may deviate slightly from the specified behaviour. The following three types of imperfection must be taken into account:

- expected values may be missing from the observations. The more values which are missed, the less confidence we will have that the observations meet the specification.
- junk values may be observed which are not expected. This will be as though extra values have been inserted into the specified behaviour and the more junk which is inserted, the less confidence we will have that the observations meet the specification.
- the observed values may not match the specified values exactly. The observations will contain characteristic types of noise, such as the classifications which are described above. The more noise which an observed value contains, the less confidence we will have that it meets the specification.

The definition of D for the body component $[\![\$p]\!]$ takes all of these imperfections into account. The sequence γ is the empty sequence and the operator J will map a set of observed sequences to a new set which contains all possible combinations of junk added to the left and right of each sequence. As more junk is added by J, the confidence in each sequence is reduced.

Given an observed sequence of values ω and a behaviour rule r then the sequence is specified by the rule with closeness c and producing the value v if

$$(\omega, v, c) \in D(r)$$

v will be the best value specified for ω when

$$\forall (\omega, v', c') \in D(r) \bullet c' \geq c$$

A meaning has been given to a behaviour specification as a set of event sequences. If an observed sequence of events is found in the set then it is said to exhibit the behaviour. Characteristic imperfections in the input data are handled by adding corresponding imperfections to the sequences generated from the specification. An input sequence may match multiple sequences in the behaviour. A confidence is attributed to a match and is the degree to which it differs from the original specification.

The meaning of a behaviour specification does not help in evaluating a recognition since the sets of event sequences will be too large to compute. The rest of this paper will describe a computational mechanism which recognises behaviour by comparing the input value sequences with the specified sequences element by element. In this way, the specified sequences only need to be developed as far as is required in order to determine whether or not the current input value matches or not. Where multiple matches are possible, a simple belief mechanism will choose between competing parses. §5 will describe a simple functional language and §6 will describe how conventional parsing techniques can be represented in this language and extended to perform behaviour recognition.

5 A functional language

The parsing of behaviours will be described using a simple functional language. The syntax of the language is given in figure 3 and the semantics will be given in appendix B. The novel feature of the language is that it has been enriched with a builtin operator for non-determinism, *fork*, and a construct, **die**, which causes the current computation to commit suicide. The language is divided into: the core, which is given a semantics by a finite state transition machine, and the rest which is given a semantics by a translation to the core.

The syntactic categories are: expressions E; identifiers I; numbers N; strings S; patterns P; data constructors K; case expressions arms A; infix operators O; bindings B; and toplevel definitions T. X^+ means a sequence of one or more constructs of syntactic category X. The core syntax is given by C which is essentially the λ-calculus, the rest of this section will briefly describe the translation of expressions in E to expressions in C, if $e_1 \in E$ and $e_2 \in C \cup E$ then we will represent the translation as $e_1 \longrightarrow e_2$ where e_1 and e_2 are equivalent expressions.

E supports curried functions so

$$\lambda p_1 p_2 \ldots p_n.e \longrightarrow \lambda p_1.\lambda p_2.\ldots\lambda p_n.e$$

A pattern may occur in any binding position and serves to restrict the domain of the underlying function and to "destructure" the value which is bound. A pattern may be one of: _ in which case the value is ignored; $i \in I$ in which case the function is total and the value will be bound to i; $(p_1, \ldots, p_n)$ in which case the value must be an n-tuple whose components match and are destructured by the corresponding p_i; or kp in which case the value must be the result of applying the constructor k to a value which matches and is destructured by p. If any of the matching tests fail then the function will return the distinguished value ϵ. The following translations describe the semantics of patern matching:

$$\lambda\_.e \longrightarrow \lambda i.e$$

$$\begin{array}{l}\lambda(p_1,\ldots,p_n).e \longrightarrow \\ \qquad \lambda i.\ \mathbf{if}\ \mathit{isntuple}(i) \\ \qquad\qquad \mathbf{then}\ \ \mathbf{let}\ \ p_1 = i \uparrow 1\ \ \mathbf{in}\ldots \\ \qquad\qquad\qquad\quad \mathbf{let}\ \ p_n = i \uparrow n\ \ \mathbf{in}\ \ e \\ \qquad\qquad \mathbf{else}\ \ \epsilon\end{array}$$

$$\begin{array}{l}\lambda kp.e \longrightarrow \lambda i.\ \mathbf{if}\ \ \mathit{isk}(i) \\ \qquad\qquad\quad \mathbf{then}\ \ (\lambda p.e)(\mathit{stripk}(i)) \\ \qquad\qquad\quad \mathbf{else}\ \ \epsilon\end{array}$$

The identifier i is not free in the expression e. The predicate *isntuple* will test whether or not its argument is a tuple of length n, the infix operator $\_ \uparrow \_$ will extract the desired element from a tuple, the predicate *isk* tests whether or not its argument has been constructed using the constructor k and the operator *stripk* is the inverse of the constructor k. **let**- and **where**-binding are defined in terms of a translation to function application:

$$\mathbf{let}\ \ p = e_1\ \ \mathbf{in}\ \ e_2 \longrightarrow (\lambda p.e_2)e_1$$

$$\mathbf{let}\ \ pp^+ = e_1\ \ \mathbf{in}\ \ e_2 \longrightarrow (\lambda p.e_2)\lambda p^+.e_1$$

$$e_1\ \ \mathbf{where}\ \ p = e_2 \longrightarrow \mathbf{let}\ \ p = e_2\ \ \mathbf{in}\ \ e_1$$

Bindings are translated so that there is a single identifier on the left hand side and then multiple bindings are "tupled up" to produce a single binding:

$$ip^+ = e \longrightarrow i = \lambda p^+.e$$

$$(i_1 = e_1)(i_2 = e_2)\ldots(i_n = e_n) \longrightarrow$$

$$(i_1, i_2, \ldots, i_n) = (e_1, e_2, \ldots, e_n)$$

Recursive bindings are defined using the "paradoxical" combinator $\mathbf{Y}$ which is defined so that it finds the fixed point of a function *i.e.*

$$f(\mathbf{Y}(f)) = \mathbf{Y}(f)$$

A **let rec** expression is transated into a **let** expression which uses $\mathbf{Y}$ to construct the cyclic structure or recursive function:

$$
\begin{aligned}
E ::= \;& I \mid N \mid S \mid \lambda P^{+}.E \mid \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \mid \\
& (E_1, E_2, \ldots, E_n) \mid \textbf{let } P^{+} = E_1 \textbf{ in } E_2 \mid \textbf{let rec } P^{+} = E_1 \textbf{ in } E_2 \mid \\
& \textbf{die} \mid E \textbf{ where } B^{+} \mid \textbf{case } E \textbf{ of } A^{+} \textbf{ end} \mid E_1 \, O \, E_2 \mid (E) \\
P ::= \;& I \mid \_ \mid (P_1, P_2, \ldots, P_n) \mid K P \\
A ::= \;& P \Rightarrow E \\
O ::= \;& + \mid - \mid \otimes \mid \ldots \\
B ::= \;& P^{+} = E \\
T ::= \;& \textbf{let } P^{+} = E \mid \textbf{let rec } P^{+} = E \\
C ::= \;& I \mid N \mid S \mid \lambda I.C \mid \textbf{if } C_1 \textbf{ then } C_2 \textbf{ else } C_3 \\
& \mid (C_1, C_2, \ldots, C_n) \mid \textbf{die}
\end{aligned}
$$

Figure 3: A The concrete syntax of a functional language

$$\textbf{let rec } i \; p^{+} = e_1 \textbf{ in } e_2 \longrightarrow$$

$$\textbf{let } i = \mathbf{Y} \lambda i p^{+}.e_1 \textbf{ in } e_2$$

A **case** expression is translated to the application of a function. Each case arm is translated to a function which will return ϵ if the case pattern does not match the supplied value:

$$\textbf{case } e \textbf{ of } a^{*} \textbf{ end} \longrightarrow a^{*}(e)$$

$$p \Rightarrow e \longrightarrow \lambda p.e$$

$$a_1 a_2 \longrightarrow \lambda i.\textbf{if } a_1(i) = \epsilon \textbf{ then } a_2(i) \textbf{ else } a_1(i)$$

All infix operators are assumed to be curried so that:

$$e_1 \, o \, e_2 \longrightarrow o e_1 e_2$$

where application associates to the left. T defines the syntax of top level definitions.

6 Behaviour recognition as parsing

Parsing may be viewed as generating all the strings in the formal language and then testing a candidate string for set membership. Alternatively, conventional parsing techniques approximate the set of all language strings by producing portions of the strings on demand as each input token is processed. This section will describe how conventional parsing may be modified to produce a behavioural recognizer which handles the characteristics of real input data. Each of the syntactic constructors that were defined in figure 1 will be given a definition as an operator in the functional language. A behaviour specification which is constructed using the operators will *parse* the sequence of input values in a manner which is consistent with the meaning for the specification as defined by D.

6.1 Basic building blocks

Each behaviour parser is an operator which maps a package of control parameters to a new package of control parameters. Each control package has the following format,

$$(\textit{stream}, \textit{conf}, \textit{env}, \textit{vars})$$

where:

- *stream* is an input stream of observed events. The operator *read* will map an input stream to the next input event and a new input stream.
- *conf* is a package which contains the number of values which have matched and the *hyp*, *junk* and *noise* confidence values.
- *env* is an environment which maps variables to values.

```
let (!, is!, strip!) = newcnstr(1)

let (@, is@, strip@) = newcnstr(1)

let (pattern :- body) (stream, conf, env, vars) arg =
    let (value, vars) = construct(pattern, vars) in
    let (conf, env) = unify(value, arg, conf, env)
    in body(stream, conf, env, vars)

let $ pattern (stream, conf, env, vars) =
    let (value, vars) = construct(pattern, vars) in
    let (stream, conf, env) = consume(value, stream, conf, env)
    in (stream, conf, env, vars)

let & parser pattern (stream, conf, env, vars) =
    let (arg, vars) = construct(pattern, vars) in
    let (stream, conf, env, _) = parser(stream, conf, env, {}) arg
    in (stream, conf, env, vars)

let ? pattern (stream, conf, env, vars) =
    let (boolexp, vars) = construct(pattern, vars)
    in if eval(boolexp, env) then (stream, conf, env, vars) else die

let (parser_1 ; parser_2) = parser_2 ∘ parser_1

let (parser_1 | parser_2) = fork(parser_1, parser_2)
```

Figure 4: The parser building operators

- *vars* is an environment which maps variable names to variables.

The operator D, which maps behaviour specifications to their meaning, substitutes values for variable names and tags the resulting event sequences with the collection of bindings from variable names to the substituted values. When event sequences are concatenated, only those sequences where the bindings for variable names common to both sequences are the same (modulo noise). The parsers use the same technique but bind variable names to variables, which represent all the possible values which D would substitute for a variable name, and use unification to ensure that different occurrences of the same variable name are always bound to the same value (modulo noise). D will construct a set of tagged event sequences from the body of a specification and then use the head pattern to re-tag each sequence with a value. Using D the variable binding tags on a sequence can never escape from a rule so that occurrences of the same variable name in different specifications can never get confused. *env* is an environment which maps variables to values and which is updated by unification. Each rule is provided with the current value of *env* and will produce a new *env* as a result. *vars* is an environment which maps variable names to particular variables which are bound in *env*. Each time a rule is invoked, a fresh collection of variables are allocated and bound to the variable names by resetting the *vars* environment to $\{\}$. Making the value of *vars* local to a rule invocation ensures that the variables associated with variable names in a specification cannot be confused with variables associated with the same names in a different rule or in a different invocation of the same rule.

Figure 4 shows the definition of each of the basic parser building operators. The operator *newcnstr* will produce a new data constructor, predicate and stripper. The operator *construct* will construct a value using a pattern and the current value of *vars* by substituting variables for each occurrence of !*varname*. The association between the variable name and the variable for each invocation of a parser will be done lazily, *i.e.* when a variable name is looked up in *vars* if it does not exist then it is added. This is why *construct* produces a value and an updated *vars* environment. *unify* is a standard unifier [13] which has been modified to cope with noisy data. *eval* will evaluate an expression which is in the form of a term. The definitions of *unify*, *construct* and *eval* are given in appendix A.

The :- operator constructs a value from the head pattern and then unifies it with the supplied argument. If this succeeds then the modified package of control parameters is supplied to the body. The $ operator will construct a value from the token pattern and then attempt to match the next input token against the value using *consume* which is defined in section 6.2. The & operator will construct an argument and then supply it to the named parser. The ? operator will construct the expression term and then evaluate it, if the result is true then the result is a modified package of control parameters otherwise the parse fails by evaluating the expression **die**. Parsers are composed sequentially using _ ; _ which corresponds to function composition. The _ | _ operator constructs parsers which succeed when either of the two arguments succeed (including both). The *fork* operator, which is fully described in §8, builds a new parser which will perform both parses independently.

6.2 Terminals

An input token is consumed in response to parsing a terminal of the grammar. The operator $ introduces terminals and figure 4 shows its definition using the *consume* operator. *consume* will map a package of control parameters and a terminal to a new package such that the next input token has been consumed and matched against the terminal. *consume* must handle missing input and input which contains junk. This section shows how *consume* is composed of three sub-operators which correspond to matching, hypothesizing and ignoring input tokens.

Figure 5 shows the definition of the operators involved in consuming a terminal and affecting the confidence in the current parse. The *conf* package which is passed to and returned from each parser will have the following format

$$(m, n, h, j, c)$$

where m is the number of exact matches which have taken place, n is a measure of the amount of noise which has been detected in the matches, h is the number of expected input tokens which have been hypothesized, j is the number of input tokens which have been considered junk and c is the most recently registered confidence level. As a parse proceeds, the values in the *conf* package will be affected and at strategic times, the parse may update its confidence level by combining the individual values into a single number and applying the operator *setconf*. There is no fixed way to combine the confidence values but figure 5 suggests a definition for the operator *combineconf*. The individual components of the *conf* package are updated by the operators *addmatched*, *addnoise*, *addhyp* and *addjunk*. When a parse wishes to update its current confidence level then it uses the operator *updateconf*.

The operator *consume* will attempt to match the next input token against a terminal of the gram-

let $\mathit{combineconf}(m, n, h, j) = m^2 - (n + h + j)$

let $\mathit{addmatched}(m, n, h, j, c) = (m + 1, n, h, j, c)$

let $\mathit{addnoise}((m, n_1, h, j, c), n_2) = (m, n_1 + n_2, h, j, c)$

let $\mathit{addhyp}(m, n, h, j, c) = (m, n, h + 1, j, c)$

let $\mathit{addjunk}(m, n, h, j, c) = (m, n, h, j + 1, c)$

let $\mathit{updateconf}(m, n, h, j, c) = (m, n, h, j, \mathit{setconf}(\mathit{combineconf}(m, n, h, j)))$

```
let rec consume = fork(fork(insert, missing), match)
        where
            match(terminal, stream, conf, env) =
                let (stream, token) = read(stream) in
                let (conf, env) = unify(terminal, token, conf, env)
                in (stream, updateconf(addmatched(conf)), env)

            missing(terminal, stream, conf, env) =
                let (conf, env) = unify(terminal, default, conf, env)
                in (stream, updateconf(addhyp(conf)), env)

            insert(terminal, stream, conf, env) =
                let (stream, _) = read(stream)
                in consume(terminal, stream, (updateconf(addjunk(conf))), env)
```

Figure 5: The definition of *consume*

mar. Since the input will not be perfect, the match must take into account noise, junk and missing information. *consume* is defined using three operators *match*, *missing* and *insert* by combining them using the *fork* operator. When *consume* is applied to a terminal, a stream, a confidence package and a variable environment, three new parsers are "forked" and will continue independently. The first parser will attempt to match the terminal against the next input token. The match is performed by unifying the expected tokens against the next input token, where the unifier will take into account noisy matches. If the match fails then the parser will commit suicide. The second parser will assume that the expected value is missing from the input stream and will hallucinate it by matching the expected token against some default value. The third parser will assume that the next input token is junk and will consume it and throw it away before calling *consume* again. In each case the parser will update its confidence value accordingly.

7 An example behaviour specification

This section will give an example behaviour specification and two event sequences which will contain the characteristic features of real world data whilst exhibiting the behaviour. Aircraft are serviced on the ground at airports between flights. Servicing activities include refuelling, restocking the galley and loading or unloading baggage. Each activity is performed by a particular class of vehicle which will perform a reasonably simple prescribed task, for example a baggage load will be performed by a fork lift truck driving up to either the forward or aft hold of the aircraft, loading the baggage and then driving away.

A simple declarative language has been developed in order to represent behaviour specifications. Figure 6 shows an example behaviour specification which describes the loading of baggage by a fork lift truck at either the fore-hold or the aft-hold.

baggageload specifies sequences of tokens which represent a fork lift truck entering the scene, loading baggage onto an aircraft and then leaving the scene. *baggageload* is constructed from three sub-specifications (only two of which are shown). *load* specifies the sequences of event tokens which represent a fork lift truck loading baggage at either the fore- or aft- holds. *loadafthold* specifies sequences of tokens which represent a loading behaviour at the aft-hold. *loadforehold* is not defined but it is the same as *loadafthold* with *fore* substituted for *aft*.

The *loadafthold* specification represents pairs of event tokens such that the first token signals a fork lift truck arriving at the aft-hold and the second token signals the same truck leaving the aft-hold. The specification is parameterized by two variables !"*object*" and !"*wait*". The first variable forces the same fork lift truck to arrive at and leave from the hold. The second variable represents the minimum amount of time for baggage to be loaded into the hold.

The *load* specification represents pairs of event tokens which are consistent with either the *loadafthold* or the *loadforehold* specifications. The values of the variables !"*object*" and !"*wait*" which are supplied to *load* are passed on to both of the sub-specifications.

The *baggageload* specification represents sequences of four tokens such that the first token signals the arrival of a fork lift truck in the scene, the next two tokens are specified by *load* and the final token signals the fork lift truck leaving the scene. The variable !"*object*" forces all the tokens to represent the behaviour of a single fork lift truck. The variable !"*object*" is viewed as a synthesized attribute of the grammar since the sequence of observed input tokens will supply its value. The variable !"*wait*" is viewed as an inherited attribute of the grammar since its value is supplied when the *baggageload* specification is used.

Figure 7 shows two sequences of input tokens which will be recognized as *baggageload* with a waiting time of 10 minutes. The sequences have examples of the three characteristic features of real input data. The classification of the entity in the first sequence is initially "*vehicle*" then "*truck*" and then "*forklift*". The first sequence of tokens contains an inserted event which describes the fork lift truck turning through θ degrees. finally, an expected event token is missing from the second sequence.

If a behaviour specification is viewed as a set of event sequences then *baggageload* must contain both of the sequences described in figure 7.

8 Control

Behaviours for single entities are recognized by parsing a stream of input tokens using the parser developed in §6. Such a parser uses the non-deterministic operator, *fork*, and a computational suicide pill, **die**, to split parses at choice points and to drop parses which have failed. Each parse is associated with a confidence level which describes how closely the input tokens have matched the expected behaviour. The parser is tolerant of noise, missing elements and inserted elements but is capable of hallucinating a parse where none exists, to the extent that all input tokens are ignored and all

```
let (token, istoken, striptoken) = newcnstr(4)

let loadafthold =
    (!("object"), !("wait")) :-
        $token ("arrivesatah", !("object"), @("forklift"), !("t1"));
        $token ("leavesfh", !("object"), @("forklift"), !("t2"));
        ?<(+(!("t1"), !("wait")), !("t2"))

let load =
    (!("object"), !("wait")) :-
        &loadforehold(!("object"), !("wait")) |
        &loadafthold(!("object"), !("wait"))

let baggageload =
    (!("object"), !("wait")) :-
        $token ("arrives", !("object"), @("forklift"), !("t1"));
        &load(!("object"), !("wait"));
        $token ("leaves", !("object"), @("forklift"), !("t2"))
```

Figure 6: An example behaviour specification

$token(\text{"}arrives\text{"}, \text{"}o_1\text{"}, @(\text{"}vehicle\text{"}), 14)$	$token(\text{"}arrives\text{"}, \text{"}o_2\text{"}, @(\text{"}forklift\text{"}), 36)$
$token(\text{"}arrivesatfh\text{"}, \text{"}o_1\text{"}, @(\text{"}truck\text{"}), 15)$	$token(\text{"}arrivesatah\text{"}, \text{"}o_2\text{"}, @(\text{"}forklift\text{"}), 37)$
$token(\text{"}leavesfh\text{"}, \text{"}o_1\text{"}, @(\text{"}forklift\text{"}), 26)$	
$token(\text{"}turns(\theta)\text{"}, \text{"}o_1\text{"}, @(\text{"}forklift\text{"}), 28)$	$token(\text{"}leaves\text{"}, \text{"}o_2\text{"}, @(\text{"}forklift\text{"}), 51)$
$token(\text{"}leaves\text{"}, \text{"}o_1\text{"}, @(\text{"}forklift\text{"}), 30)$	

Figure 7: Tokens for *baggageload*

the expected terminals are hypothesized. Such a parse will have a very low confidence level associated with it. The work which is involved in parsing a behaviour can be greatly reduced if the parsers are controlled in terms of the respective confidence levels. For some applications it may be sufficient to develop the single parse with the highest confidence level whereas for other applications it may be necessary to develop a collection of parses which are constructed by constantly reviewing their confidence levels.

This section will show how the parses are controlled by giving an operational semantics to the implementation language using a transition machine. When a non-deterministic operator (constructed using *fork*) is applied to a value, a pair of machines are "spawned". Each machine will execute a different parse thread and will contain its own confidence level. The machines are controlled by a program which can decide whether to continue executing the machine based on its current confidence level.

The SECD [15] machine gives an operational semantics to the λ-calculus. In [10] the SECD machine is extended with primitives for non-determinstic computation. We go further and define an IPARG[2] machine which includes primitives for non-determinism but also includes control information which represents the current level of confidence in the computation.

Like the SECD machine, the IPARG machine evaluates by performing state transitions. Unlike the SECD machine, the IPARG state transition function will map a single state to a set of states, each of which arises because of the non-determinism in the program. Unlike the SECD machine, the IPARG machine has a state component which contains a value representing the current level of confidence in the computation. The program will update this component using the builtin operator *setconf* and the confidence value will be used in ranking the current collection of computations in order of interest. The IPARG state transition function is defined in figure 10

The non-determinism in a computation arises from the builtin operator *fork*. The operand of *fork* is a pair of operators, o_1 and o_2, and the result of the application is a new operator which when applied to an operand will non-deterministically apply either o_1 or o_2. The definition of the builtin operator *fork* is given in figure 10.

Each time an evaluating program performs an action which causes its confidence level to change, it will use the builtin operator *setconf* to update the state component to the new value. The confidence level is used to rank the computations in order of interest, so a program may make itself more or less interesting by affecting this value. The definition of *setconf* is given in figure 10.

Suppose that the IPARG machine is implemented in the functional language and that there is a constructor

$$(\_,\_,\_,\_,\_)$$

for machine states and a function $\longmapsto$ which maps a single machine state to a set of machine states which is the result of performing a single non-deterministic atomic machine transition. Sets of machine states will be implemented as lists with the following constructors: $[]$ as the empty list and $::$ is the list constructor which adds an element x to a list l to produce $x :: l$. We will now show how three different control strategies can be implemented for the IPARG machine depending upon the resources which are available for the parser.

Figure 8 gives the definitions of three operators $exec_1$, $exec_2$ and $exec_3$. These operators will each map a set of IPARG initial machine states to a set of final machine states and differ in the ways in which they control the machines. The other operators in figure 8 are used in the control operators and are described as follows: the operator $\backslash$ is used to construct a list homomorphism from a right associative binary operator $\otimes$, a unary operator f and a base value b, given a list

$$x_1 :: (x_2 :: [])$$

the homomorphism

$$\backslash\ (\otimes)\ f\ b$$

will produce the value

$$f(x_1) \otimes f(x_2) \otimes b$$

the operator $/$ is used to remove values which do not satisfy a given predicate p from a list l; $\mathbf{I}$ is the identity operator; the operator $++$ will append two lists; the operator $\sim$ is used to negate a predicate; the operator *conf* maps an IPARG machine state to its current confidence value which is a number; the predicate *done* is true of an IPARG machine state when there is no more computation to be performed; the predicate *less* is true for a value n and a machine state s when the confidence value for s is less than n; the operator *evals* will map a set of IPARG machine states to the set of states which is produced by performing a single machine transition for each starting state and then flattening the resulting lists.

The operator $exec_1$ will map a set of initial machine states to the set of terminal states which is constructed by running all of the states to completion.

[2] The name IPARG is associated with the system described in this paper.

$$
\begin{array}{l}
\textbf{let rec} \ \backslash \ (\otimes) \ f \ b \ l = \\
\quad\quad \textbf{case} \ l \ \textbf{of} \\
\quad\quad\quad\quad [] \Rightarrow b \\
\quad\quad\quad\quad x :: l \Rightarrow (f(x)) \otimes (\backslash \ (\otimes) \ f \ b \ l) \\
\quad\quad \textbf{end} \\
\\
\textbf{let rec} \ / \ p \ l = \\
\quad\quad \textbf{case} \ l \ \textbf{of} \\
\quad\quad\quad\quad [] \Rightarrow [] \\
\quad\quad\quad\quad x :: l \Rightarrow \\
\quad\quad\quad\quad\quad \textbf{if} \ p(x) \\
\quad\quad\quad\quad\quad \textbf{then} \ x :: (/ \ p \ l) \\
\quad\quad\quad\quad\quad \textbf{else} \ / \ p \ l \\
\quad\quad \textbf{end} \\
\\
\textbf{let} \ \mathbf{I}(x) = x \\
\\
\textbf{let} \ x \, max \, y = \textbf{if} \ x > y \ \textbf{then} \ x \ \textbf{else} \ y \\
\\
\textbf{let} \ l_1 +\!\!+ \, l_2 = \backslash \ (::) \ \mathbf{I} \ l_2 \ l_2 \\
\\
\textbf{let} \ \sim p \ x = \textbf{if} \ p(x) \ \textbf{then} \ \mathit{false} \ \textbf{else} \ \mathit{true} \\
\\
\textbf{let} \ \mathit{conf}(\_,\_,\_,n,\_) = n \\
\\
\textbf{let} \ \mathit{done}(\_,\_,s,\_,d) = (s = []) \,\&(d = ()) \\
\\
\textbf{let} \ \mathit{less} \ n = (< n) \circ \mathit{conf} \\
\\
\textbf{let} \ \mathit{evals} = \backslash \ (+\!\!+) \ (\longmapsto) \ [] \\
\\
\textbf{let rec} \ \mathit{exec}_1 \ s = (/\mathit{done} \ s) +\!\!+ (\mathit{exec}_1(\mathit{evals}(/(\sim \mathit{done})s))) \\
\\
\textbf{let rec} \ \mathit{exec}_2 \ n \ s = (/\mathit{done} \ s) +\!\!+ (\mathit{exec}_2 \ n \ (/(\mathit{less} \ n)(\mathit{evals}(/(\sim \mathit{done})s)))) \\
\\
\textbf{let rec} \ \mathit{exec}_3 \ s = (/\mathit{done} \ s) +\!\!+ (\mathit{exec}_3((/(\mathit{less} \ n)\mathit{notdone}) +\!\!+ (\mathit{evals}(/(\sim (\mathit{less} \ n))\mathit{notdone})))) \\
\quad\quad\quad\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad\quad\quad\quad\quad n = \backslash \ (\mathit{max}) \ \mathit{conf} \ 0 \ s \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{notdone} = /(\sim \mathit{done})s
\end{array}
$$

Figure 8: Definitions of controllers

This control operator does not take into account the confidence values of any of the machine states and may take a great deal of resources to find the desired parse.

The operator $exec_2$ will map a confidence level n and set of initial machine states to a set of terminal machine states s such that each of the states in s has a confidence level which is greater than n. This control operator will ignore any parses whose confidence levels drop below the threshold value n. $exec_2$ will use less resources than $exec_1$ because it will not develop any parses which are not required. Unfortunately $exec_2$ is not perfect because it may throw away some of the parses which are desired if they dip temporarily below the threshold value.

The operator $exec_3$ will map a set of initial machine states to a set of machine states, some of which have terminated. At any stage, $exec_3$ will only develop those parses which have the highest confidence levels. Parses which are initially found interesting will be developed until they eventually succeed or until their confidence levels drop below previously "frozen" parses. If all of the parses have the same confidence levels throughout, then $exec_3$ will consume the same amount of resources as $exec_1$. If the confidence level for the "correct" parse becomes the highest very quickly and stays that way, then $exec_3$ will consume only slightly more resources than $exec_2$.

The three control operators which are described above, are not the only control mechanisms for the IPARG machine. The best controller is likely to depend upon the characteristics of the parsing for the type of application.

9 Conclusion, Discussion and Further Work

Behavioural specifications of single entities have been described as attribute grammars and behavioural recognition as parsing. A behaviour specification generates a formal language which is a set of sequences of event tokens. The sentences in the formal language are characterised by missing, inserted and noisy data. A conventional parser has been modified to deal with the characteristic features of realistic input data. Such a parser is controlled with respect to a confidence level which describes the difference between the token sequences which were expected and those that were received.

The parsers are given a semantics using a simple functional programming language which has been extended with constructs for non-determinism. The operational semantics of the functional language is defined by the IPARG machine which is constructed from the SECD machine by enriching it with primitives for non-deterministic computation and including a confidence level with each computation. Each computation may use a primitive operation to change its confidence level and therefore make itself more or less interesting. The machine transition function maps a single state to a list of states each of which continues independently. An entire program evaluation is performed by the function *exec* which uses the confidence levels to rank computations. Different variations of the *exec* operator may be defined which give rise to different strategies for controlling the non-deterministic program.

We have not shown a formal proof of the consistency and completeness of the parsing operators with respect to the semantics of behaviour specifications as defined by D in figure 2. In general, a parser will not be complete, *i.e.* not produce the best value for a parse, because the definition of *exec* will have been designed to work within specific resource limits.

The system described in this paper has been implemented as part of the VIEWS ESPRIT project. The implementation is written in Common Lisp and has been used to construct behaviour recognisers which describe a realistic scene in which baggage is loaded on and off an aircraft, the galley is restocked, the toilets are serviced, the aircraft is refuelled and passengers embark and disembark.

We believe this work to be a successful synthesis of many related threads of pattern recognition – parsing attribute grammars with generalised constraints, noise related errors and heuristics, using state-of-the-art techniques. The novel implementation mechanism provides a particularly clean and easily controlled vehicle for experimentation with different parsing techniques. In particular, high level functional programs can be freely mixed with grammar rules and the interpretive mechanism for the language supports features which are ideally suited to pattern recognition and need not clutter the recognition program.

In comparison to related work, our approach allows missing, inserted and noisy data to be handled in a generic and modular fashion which is not true of conventional attribute parsing techniques which must extend each production with mechanisms for handling errors – an approach which is also supported. The nature of conventional backtracking parsers is such that if the search space is developed in a depth first fashion and missing input tokens are hypothesised then more appropriate alternative parses may never be developed. Our approach is related to stochastic grammars in that confidence levels are associated with alternative parses, but unlike these grammars we are able to have very flexible control by programming the IPARG machine. Our approach is related to attribute grammars since each input token has attributes and production rules synthesise attributes. Unlike conventional attribute grammars, the attributes are unified and general constraints are applied to the attributes at all levels of the grammar. This provides a very expressive language for expressing behaviours. The unification technique has much in common with parsing using lazy functional programming languages and it would be interesting to compare these approaches in detail. Combinators have been used to construct parsers but our approach extends existing systems with components for handling characteristic errors and confidence levels.

In our experiments within the VIEWS project we have found the prototype parser to be of satisfactory performance with respect to simple scenes. Since the parser control can be programmed, as as scene complexity increases, adding extra cleverness to the control will maintain the desired performance. Since each parser combinator has a standard interface, which is a package of information, new combinators can easily be invented and the package can be extended with information which can be used to control the parse. A feature which we expect to be of importance with parsers of this kind is a "cut" mechanism (rather like that of Prolog) which can be inserted into the grammar rules in order to discard unwanted alternative parses. The approach described here will parse strings of input tokens; it would be interesting to see if the approach could be generalised for trees and graphs.

10 Acknowledgements

The author would like to acknowledge the support of the Commission of European Communities and Marconi Radar and Control Systems for funding of this work, carried out largely under Project 2152: VIEWS, Visual Interpretation and Evaluation of Wide-area Scenes. The design and implementation of the behaviour recognition system described in this paper was performed by A. N. Clark and A. G. Hill. The author wishes to thank A. G. Hill and the two anonymous reviewers for helpful comments on earlier drafts of this paper.

References

[1] Bunke, H. (1982) Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **PAMI-4**, 6.

[2] Burge, W. (1975) *Recursive Programming Techniques.* Addison-Wesley.

[3] Corrall, D. R. and Hill, A. G. (1992) Stand Area Surveillance at Airports For Decision Support. *ASTAIR conference.*

[4] Corrall D. R. and Hill A. G. (1992) Visual Surveillance. *GEC Review* **8**, *1.*

[5] Corrall, D. R. Clark, A. N. and Hill, A. G. (1993) Airside Ground Movements Surveillance. *Agard Conference Proc. 538: Machine Intelligence in Air Traffic Management.*

[6] Flasinski, M. (1993) On the Parsing of Deterministic Graph Languages for Syntactic Pattern Recognition. *Pattern Recognition. 26*, 1.

[7] Flick, T. E., Jones, L. K. (1986) A Combinatorial Approach for Classification of Patterns with Missing Information and Random Orientation. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **PAMI-8**, 4.

[8] Frost, R. A. (1992) Constructing Programs as Executable Attribute Grammars. *The Computer Journal.* **35**, 4.

[9] Fu, K. S. (1974) *Syntactic Methods in Pattern Recognition.* Academic Press.

[10] Henderson, P. (1980) *Functional Programming Applications and Implementation.* Prentice-Hall International.

[11] Hutton, G. (1989) Parsing Using Combinators. *Proc. Glasgow Workshop on Functional Programming.* Springer Workshops in Computing.

[12] Johnsson, T. (1987) Attribute Grammars as a Functional Programming Paradigm. *LNCS 274 Springer-Verlag.*

[13] Knight, K. (1989) Unification: A Multidisciplinary Survey. *ACM Computing Surveys.* **21**, 1.

[14] Knuth, D. E. (1968) Semantics of context-free languages. *Mathematical Systems Theory Journal.* **2**, 2.

[15] Landin, P. J. (1964) The Mechanical evaluation Of Expressions. *The Computer Journal* **6**.

[16] Landin, P. J. (1966) The Next 700 Programming Languages. *Communications of the ACM.* **9**, 3.

[17] Nagel, H. H. (1988) From Image Sequences Towards Conceptual Descriptions. *Image and Vision Computing.* **6**, 2.

[18] Partridge, A., Wright, D. (1994) Parser combinators need four values to report errors. *University of Tasmania, Computer Science Report.*

[19] Schank, R. C. and Abelson, R. P. (1977) *Scripts, Plans, Goals and Understanding.* Hillsdale New Jersey. Erlbaum.

[20] Tang, G. Y., Huang, T. S. Huang (1979) A Syntactic-Semantic Approach to Image Understanding and Creation. *IEEE Trans. on Pattern Analysis and Machine Intelligence.* **PAMI-1**, 2.

[21] Wadler, P. (1990) Comprehending Monads. *In Proc 19th Symp. on Lisp and Functional Programming.* Nice, ACM.

[22] Woods, W. A. (1977) Shortfall and Density Scoring Strategies for Speech Understanding Control. *IJCAI-77.*

A Utilities

Figure 9 gives the definitions of the unification, term construction and expression evaluation operators which have been used in this paper.

The operator *unify* is a standard unification algorithm which has been extended to cope with noisy classification matches. If the unifier fails to match the two input values then the final arm of the **case** expression in*unify* will cause the current parse to commit suicide. The operator *zip* will map two lists of equal length to a list of pairs and the operator *unifypairs* will succeed when each pair of values in the list unify. Note that the operator $\_ \bullet \_$ will look a value up in an environment and *dom* will map an environment to the set of things which may be looked up in it.

The operator *construct* will reconstruct the value v by replacing each variable of the form $!s$ by the corresponding variable which is bound to the name s in e. If no variable exists then a fresh variable is constructed using the operator *newvar*.

The operator *eval* will evaluate an expression which is supplied in the form of a value v.

B Transitions

The parsers which are developed in the main text are written in a functional language. This language is given an operational semantics by describing how it evaluates using a transition machine. The machine is defined in figure 10 and consists of a set of states and a transition function which is defined to map single machine states to sets of machine states. Each machine state has the following form

$$(s, e, c, x, d)$$

where s is a list of program outcomes, e is an environment which associates program identifier names and their current values, c is a list of program expressions and machine instructions, x is a value which represents the current confidence in the parse which is being performed by the machine and d is a value which is either empty () or a state

$$(s, e, c, d)$$

without an x component. The letter k stands for any constant such as a string or a number; the letter v stands for any program outcome; the letter f stands for any operator (*i.e.* applicable value). The transition function defines the atomic computational steps when a functional program is performed. Since the program may apply a fork operator, the result of each machine transition will be a set of machine states. A function "closure" is represented as a term

$$< i, e_1, e_2 >$$

where i is the formal parameter, e_1 is the environment for the closure and e_2 is the body of the closure. Machine instructions are created as the program is performed and are the terms *tup*(n), for tuple creation, @ for function application and *test*(e_1, e_2) for conditional expressions where e_1 is the consequent and e_2 is the alternative. Builtin operator are represented as terms *bif*(s) where s is the name of the operator. The builtin operators are defined by giving the transition rule which will describe what happens when it is applied. Notice that the x component of the machine is "global" *i.e.* it is not saved and restored on function application and return.

```
let rec deref(v, e) =
    if isvar(v)
    then
        if v ∈ dom(e)
        then deref(e • v)
        else v
    else v

let rec unify(v1, v2, conf, e) =
    let (v1, v2) = (deref(v1, e), deref(v2, e))
    in  case (v1, v2) of
            (!_, _) ⇒ (conf, e ⊕ (v1 ↦ v2))
            (_, !_) ⇒ (conf, e ⊕ (v2 ↦ v1))
            (v, v) ⇒ (conf, e)
            ((x1, ..., xn), (y1, ..., yn) ⇒ unifypairs(zip(v1, v2)) conf e
            (@s1, @s2) ⇒
                if glb(s1, s2) = ⊥
                then die
                else (addnoise(conf, diff(s1, s2)), e)
            ...
            _ ⇒ die
        end

let rec construct(v, e) =
    case v of
        constant(k) ⇒ (k, e)
        !s ⇒
            if s ∈ dom(e)
            then (e •s, e)
            else (v, e ⊕ (n ↦!(v)))
                where v = newvar s
        ...
    end

let rec eval(v, e) =
    case deref(v, e) of
        constant(k) ⇒ k
        !v ⇒ deref(v, e)
        +(e1, e2) ⇒
            let v1 = eval(e1, e) and
                v2 = eval(e2, e)
            in v1 + v2
        ...
    end
```

Figure 9: Utility operators

$$(s, e, i :: c, x, d) \longmapsto \{((e \bullet i) :: s, e, c, x, d)\}$$

$$(s, e, k :: c, x, d) \longmapsto \{(k :: s, e, c, x, d)\}$$

$$(s, e_1, [\![\lambda i.e_2]\!] :: c, x, d) \longmapsto \{(< i, e_1, e_2 >:: s, e, c, x, d)\}$$

$$(s, e, [\![e_1 e_2]\!] :: c, x, d) \longmapsto \{(s, e, e_2 :: e_1 :: \ @ \ :: c, x, d)\}$$

$$(< i, e_1, e_2 >:: v :: s, e_3, \ @ \ :: c, x, d) \longmapsto \{([], e_1 \oplus (i \mapsto v), [e_2], x, (s, e_3, c, d))\}$$

$$(s, e, [\![(e_1, \ldots, e_n)]\!] :: c, x, d) \longmapsto \{(s, e, e_1 :: \ldots :: e_n :: \mathit{tup}(n) :: c, x, d)\}$$

$$(v_n :: \ldots v_1 :: s, e, \mathit{tup}(n) :: c, x, d) \longmapsto \{((v_1, \ldots, v_n) :: s, e, c, x, d)\}$$

$$(s, e, [\![\textbf{if} \ \ e_1 \ \ \textbf{then} \ \ e_2 \ \ \textbf{else} \ \ e_3]\!] :: c, x, d) \longmapsto \{(s, e, e_1 :: \mathit{test}(e_2, e_3) :: c, x, d)\}$$

$$(\mathit{true} :: s, e, \mathit{test}(e_1, e_2) :: c, x, d) \longmapsto \{(s, e, e_1 :: c, x, d)\}$$

$$(\mathit{false} :: s, e, \mathit{test}(e_1, e_2) :: c, x, d) \longmapsto \{(s, e, e_2 :: c, x, d)\}$$

$$(s, e, [\![\textbf{die}]\!] :: c, x, d) \longmapsto \{\}$$

$$(\mathit{bif}(\text{“}\mathit{fork}\text{”}) :: (f_1, f_2) :: s, e, \ @ \ :: c, x, d) \longmapsto \{(\mathit{fork}(f_1, f_2) :: s, e, c, x, d)\}$$

$$(\mathit{bif}(\text{“}\mathit{getconf}\text{”}) :: () :: s, e, \ @ \ :: c, x, d) \longmapsto (x :: s, e, c, x, d)$$

$$(\mathit{bif}(\text{“}\mathit{setconf}\text{”}) :: x_1 :: s, e, \ @ \ :: c, x_2, d) \longmapsto (x_1 :: s, e, c, x_1, d)$$

$$(\mathit{fork}(f_1, f_2) :: v :: s, e, \ @ \ :: c, x, d) \longmapsto \{(f_1 :: v :: s, e, \ @ \ :: c, x, d), (f_2 :: v :: s, e, \ @ \ :: c, x, d)\}$$

$$(v :: s', e', [], x, (s, e, c, d)) \longmapsto \{(v :: s, e, c, x, d)\}$$

Figure 10: The IPARG machine