Language Factories and Modular Interpreters

Tony Clark

June 16, 2010

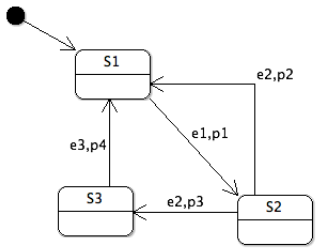

Figure 1: State Machine

1 Introduction

Languages have a syntax and semantics. Often, when we talk of a language, for example a state-machine language, an expression language or an action language, we might mean a *particular* syntax and semantics, and at other times we might want the named language to denote a *family* of *related* languages. For example, the term 'state-machine' might be used in the context of deterministic machines whose nodes are labelled with simple states and whose edges are labelled with simple events, or 'state-machine' might be used to denote any element of a set of possible languages including deterministic machines, non-deterministic machines, priority-driven machines, action-based machines *etc.*

Most language definition systems do not support language families because the syntax and semantics of any language is fixed. A *Language Factory* approach is different because it aims to capture a family in a single definition so that individual languages in the family can be instantiated from the definition. Think of a language factory as a kind of template or language pattern. This approach is motivated by the aim of creating libraries of language module definitions so that new languages can be created by instantiating a composing modules.

2 A State-Machine Factory

Consider figure 1 that shows a state machine with 3 states (labelled S1,S2,S3) and 4 transitions. The labels on the transitions are pairs of events (e1,e2,e3) and tags (p1,p2,p3,p4). The idea is that the tags can be used to encode extra information on the transitions - some interpretations of the state machine might use the tags and some might not. Therefore, the syntax of the machine in figure 1 is actually a family of syntaxes that differ in terms of whether labels are allowed or not. Each element of the syntax family defines its own well-formedness rules.

Each element of the family must define a semantics. The semantics of each element will have common features: the machine always starts in S1; the machine is event-driven; an event causes a state change defined by the outgoing event labels on the transitions from the node labelled with the current state. Here are a few possible semantic options:

- The machine is deterministic: each outgoing transition from a state must have a unique event label.

- The machine is single-threaded: some strategy is used to select one amongst multiple outgoing transitions with the same event label (variations include the first, the last, random choice, prioritization via transition tags).

- The machine is non-determinstic (multi-threaded): all possible transitions with the same label are chosen causing the machine to be in one of any number of states at any given time.

- The machine counts steps: a (single- or multi-threaded) machine keeps track of both its current state and the number of transitions it has performed.

The rest of this article describes how to encode all of these choices using XPL using a Language Factories approach. Syntax families are implemented by parameterising over the concrete syntax of a state machine. Semantic families are implemented by defining a modular interpreter for a state machine. The modular interpreter is written using a monadic-style that abstracts the key interpretive steps of machine execution as a standardized package of machine operations. The semantic variations are implemented by supplying different packages of machine operations.

3 Syntax Factories

Consider the representation of a simple state-machine:

```
{states = ['S1','S2','S3'];
 trans = [
   {source='S1';event='e1';tag='';target='S2'},
   {source='S2';event='e2';tag='';target='S1'},
   {source='S2';event='e2';tag='';target='S3'},
   {source='S3';event='e3';tag='';target='S1'}]}
```

This representation is the *abstract syntax* of a state-machine. We would like a more user-friendly style that hides the use of records and lists to represent the machine. This is achieved in XPL using a grammar definition. The grammar will process a representation as follows:

```
machine[S1,S2,S3] {
  S1-(e1)->S2
  S2-(e2)->S1
  S2-(e2)->S3
  S3-(e3)->S1
}
```

However, we want the syntax definition for state-machines to allow for a family of languages. Each member of the family differs in terms of the extra syntax used to represent transition labels. Therefore, we define the syntax of a state-machine as a function that maps a grammar of *tags* to a grammar of machines:

```
machine(tag) = {
  machine  -> 'machine' ss=states '{' t=trans* '}' {
    [| { states = ${listExp(ss)};
         trans  = ${listExp(t)}} |]
  };
  states   -> '[' s=name ss=(',' name)* ']' {
    map(fun(s) s.lift(),s:ss)
  };
  trans    -> s=name e=event t=name x=tag {
    [| { source = ${s.lift()};
         event  = ${e.lift()};
         target = ${t.lift()};
         tag    = ${x}} |]
  };
  event   -> '-(' n=name ')->' { n }
}
```

The function `machine` maps a grammar `tag` to a grammar that parses machines. The rule `machine` processes states, transitions and synthesizes a record containing fields `states` and `trans`. The function `listExp` transforms a sequence of expressions to a list expression. The rule `states` recognizes a sequence of names and returns a sequence of string expressions. The rule `trans` recognizes a source name, an event tag, a target name and a tag, and then synthesizes a record with named fields whose values are the appropriate values. Note that the tag rule is supplied as an argument. The event rule recognizes and returns an event name.

The `machine` grammar synthesizes XPL expressions. An expression is constructed using quasi-quotes [| and |] that surround concrete XPL syntax. Instead is parsing an evaluating the expression, the quotes cause the expression to be parsed and returns as a value of type *abstract-syntax*. Within the quasi-quotes, the drop quotes ${ and } surround expressions that are parsed and evaluated as usual. Therefore, `[| x + ${y} |]` constructs a binary expression whose operator is `+`, whose left hand operand

is the variable `x` and whose right hand operand is the value of variable `y` (which should be of type abstract-syntax). All values may be sent a message `lift()` that translates the receiver into an expression that recreates the receiver when the expression is evaluated. The operstor `listExp([e1,e2,...,en])` maps a sequence of expressions `[e1,e2,...,en]` to an expression `[| [e1,e2,...,en] |]`.

The `machine` operator is a language factory for producing state-machine grammars. The family members differ in terms of the tag syntax that is supplied as an argument to `machine`. Here are two members of the family:

```
simpleMachine = machine({
  rule -> { [| '' |] } })

prorityMachine = machine({
  rule -> 'priority' n=['0','9'] { n.lift() } })
```

The grammar `simpleMachine` ignores tags; each transition has an empty string as its tag value. The grammar `priorityMachine` expects each transition to have a keyword `priority` followed by a number in the range 0 to 9 which is returned (lifted as an expression).

Here are two examples:

```
m1 = intern simpleMachine {
  machine[S1,S2,S3] {
    S1-(e1)->S2
    S2-(e2)->S1
    S2-(e2)->S3
    S3-(e3)->S1
  }
}

m2 = intern priorityMachine {
  machine[S1,S2,S3] {
    S1-(e1)->S2 priority 1
    S2-(e2)->S1 priority 3
    S2-(e2)->S3 priority 2
    S3-(e3)->S1 priority 1
  }
}
```

4 Modular Interpreters

The semantics of state-machines can be defined in terms of transitions that are triggered in response to receiving events. As described above, there may be more than one semantics for a given state-machine. However, it turns out that many of the possible semantics have a similar pattern that involves checking the current state, matching transitions that are labelled with the next event and changing the current state of the machine. Such a semantic pattern constitutes a *modular interpreter* for state-machines.

A modular interpreter is parameterized with respect to the key steps that are required in order to run a program (in this case the program is a state-machine and a sequence of events). Different semantics can be implemented for the same machine by supplying the interpreter with different packages of step-implementations. This approach is related to the monadic-interpreters of Hudak and to monads in general.

Semantic processing of a state-machine involves the following steps:

`new`	Create and start a new machine.
`state`	Return the current state of the machine.
`trans`	Select a transition that satisfies a given predicate.
`become`	Change to a new state.
`stop`	Stop the machine.

The steps take arguments that allows them to be composed in order to create state-machine *processors*. The arguments are defined as follows:

`new(p)`	A new machine is processed by p which is a processor created by composing the steps listed below.
`state(f)`	Supply the current machine state to the function f which returns a machine processor.
`trans(p,t,a)`	Supply function t with a transition that satisfies p. If no transition satisfies

p then do a. Both a and the result of t are machine processors.

`become(s,p)` The current machine state changes to s and execution continues with processor p.

Given suitable implementations of the machine-steps listed above, the following code implements a monadic interpreter for a state-machine:

```
run(es) =
  letrec eval(es) =
    case es {
      [] -> stop;
      e:es ->
        state(fun(s)
          trans(fun(t) s = t.source and e = t.event,
                fun(t) become(t.target,eval(es)),
                eval(es)))
    }
  in new(eval(es))
```

Notice how the interpreter abstracts away from the implementation of the machine and from the mechanisms by which the transitions and current state of the machine are processed. In fact with a little bit of sugar we would get the following which I would hope you would agree is simple and captures the essential features of an interpreter for a state-machine:

```
run(es) =
  letrec eval(es) =
    case es {
      [] -> stop;
      e:es ->
        let s = state
        in let t = trans
                    where s = t.source and
                          e = t.event
           in become t.target;
              eval(es)
    }
  in let m = new in eval(es)
```

As it stands, the definition of `run` uses a fixed collection of step implementations (`new`, `state`, `trans`, `become` and `stop`). However, we want `run` to capture a family of different state-machine semantics. Therefore, run is modified in order to supply the package of step-implementations:

```
run(mpkg)(es) =
  import mpkg {
    letrec eval(es) =
      case es {
        [] -> stop;
        e:es ->
          state(fun(s)
            trans(fun(t) s = t.source and e = t.event,
                  fun(t) become(t.target,eval(es)),
                  eval(es)))
      }
    in new(eval(es))
  }
```

5 Semantics Factories

A Language Factory approach allows a family of languages to be defined and instantiated from a single pattern definition. We have set up a family of concrete syntax in section 3 for state-machines. The definition of `run` in section 4 defines a family of related semantics for state-machines. The members of the family must all implement the step operations: `new`, `state`, `trans`, `become` and `stop` and a package containing the implementation is supplied to `run`. This section shows how the different semantics outlined in section 2 are all implemented as packages of step operations.

5.1 Deterministic

A deterministic (or more accurately *single-threaded*) state-machine is in one state at any given time. In the following semantics, multiple transitions with the same event label from the same source state will cause the first transition to be selected:

```
DET(m) = {
  new(next) = next(head(m.states));
  stop(state) = state;
  trans(p,next,alt) =
    let T = filter(p,m.trans)
    in case T {
         [] -> alt;
         t:ts -> next(t)
       };
  become(state,next)(ignore) = next(state);
```

```
  state(f)(state) = (f(state))(state)
}
```

The steps are used in the following execution:

```
run(DET(m1))(['e1','e2','e1'])
=> s2
```

5.2 Non-Deterministic

A non-deterministic (or multi-threaded) state-machine can be in multiple states at any given time. This can be implemented by changing the `DET` package of steps slightly in order to return a sequence of states. The `stop` step is altered to produce a sequence of current states. The `trans` step might find a sequence of applicable transitions `T`. Instead of selecting the first (as in DET), NONDET uses all of the applicable transitions. This produces a list of state-lists that must be flattened:

```
NONDET(m) = {
  new = DET(m).new;
  stop(state) = [state];
  trans(p,next,alt) =
    let T = filter(p,m.trans)
    in case T {
         [] -> alt;
         ts -> fun(state)
           let funs = map(next,ts)
           in flatten(map(fun(f) f(state),funs))
       };
  become = DET(m).become;
  state = DET(m).state
}
```

The steps are used in the following execution:

```
run(NONDET(m1))(['e1','e2','e1'])
=> [s2,s3]
```

5.3 Counting Steps

The current state of the machine interpreter run as defined in sections 5.1 and 5.2 is a state-name. In the case of NONDET, a sequence of events is processed leading to multiple paths through the machine states. Some of these paths might terminate prematurely since the machine reaches a state for which there is no outgoing transition with the appropriate event label. Given a sequence of terminal states, it might be useful to know how many events were processed in order to reach each individual state. This can be implemented by a package of step operations that manage the interpreter state as a record: `{state=s;count=n}` where `s` is the current machine-state and `n` is the number of events that have been processed to reach `s`:

```
COUNT(m) = {
  new(next) = next({state=head(m.states);count=0});
  stop = NONDET(m).stop;
  trans = NONDET(m).trans;
  become(state,next)(previous) =
    next({state=state;count=previous.count + 1});
  state = NONDET(m).state
}
```

The steps are used in the following execution:

```
run(COUNT(m1))(['e1','e2','e1'])
=> [{state=s2;count=3},{state=s3;count=2}]
```

5.4 Priority Based

Finally, a deterministic state-machine might include multiple transitions leading from a state that all have the same event label. In order to control this we chose the first transition in section 5.1. However, we might want to compute the order dynamically, or to allow the programmer to specify the order. To do this we use a different syntax in the machine family. And order the transitions based on their priority:

```
tagLess(t1,t2) = t1.tag < t2.tag;
PRIO(m) = {
  new = DET(m).new;
  stop = DET(m).stop;
  trans(p,next,alt) =
    let trans = sort(tagLess,filter(p,m.trans))
    in case trans {
         [] -> alt;
         t:ts -> next(t)
       };
  become = DET(m).become;
  state = DET(m).state
}
```

The steps are used in the following execution:

```
run(PRIO(m2))(['e1','e2','e1'])
=> s2
```

Alternatively, changing the priority:

```
m3 = intern priorityMachine {
  machine[S1,S2,S3] {
    S1-(e1)->S2 priority 1
    S2-(e2)->S1 priority 2
    S2-(e2)->S3 priority 3
    S3-(e3)->S1 priority 1
  }
}
run(PRIO(m3))(['e1','e2','e1'])
=> s3
```

6 Conclusion

This article has described the idea of *Language Factories* as a single definition that captures the idea of a family of related languages. A *syntax factory* can be defined as a grammar that is parameterized over variation points. Where the language has an operational semantics (in the case of a state-machine that processes sequences of events), a *semantics factory* can be defined as a *modular interpreter* that uses general-purpose steps to process programs (in this case a state machine + event sequences). Particular languages can be defined using different implementations of the steps that are supplied to the interpreter.