Simulating Eilenberg Machines with a Reactive Engine: Formal Specification, Proof and Program Extraction.

Benoit Razet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Simulating Eilenberg Machines with a Reactive Engine:

Formal Specification, Proof and Program Extraction.

Benoît Razet

N° 6487

February 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6487--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Simulating Eilenberg Machines with a Reactive Engine:

Formal Specification, Proof and Program Extraction.

Benoît Razet

Thème SYM — Systèmes symboliques
Équipe-Projet SANSKRIT

Rapport de recherche n° 6487 — February 2008 — 19 pages

Abstract: Eilenberg machines have been introduced in 1974 in the field of formal language theory. They are finite automata for which the alphabet is interpreted by mathematical relations over an abstract set. They generalize many finite state machines. We consider in the present work a class of Eilenberg machines for which we provide an executable complete simulator. This program is described using the Coq specification language. The correction of the algorithm is also proved formally and mechanically verified using the Coq proof assistant. The Coq extraction code technology allows to translate the specification into executable OCaml code. The algorithm and proofs are inspired from the reactive engine of Gérard Huet.

Key-words: Automata, Eilenberg machines, reactive process, Coq

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Simulating Eilenberg Machines with a Reactive Engine:

Formal Specification, Proof and Program Extraction.

Résumé : Les machines d'Eilenberg ont été introduites en 1974 comme outil conceptuel important afin d'étudier la théorie des langages formels. Ce sont des automates finis étiquetés par des relations définies sur un ensemble quelconque de données. Cette notion de machine généralise bon nombre de machines d'états finis. Nous étudions une classe de machines d'Eilenberg pour lesquelles nous fournissons un programme qui permet de les simuler. Le programme s'inspire du moteur réactif ("reactive engine") introduit par Gérard Huet. Il est décrit à l'aide de l'outil de spécification et de preuve formelles Coq. La correction de l'algorithme est donnée avec une preuve formelle mécaniquement vérifiée. Coq fournit aussi une technologie d'extraction de programmes à partir de spécification que nous utilisons pour obtenir un programme OCaml. Ce programme issu de l'extraction permet de simuler n'importe quelle machine d'Eilenberg appartenant à la sous classe considée.

Mots-clés : Automates, machines d'Eilenberg, processus réactif, Coq

1 Introduction

The style of the report is inspired from the literate programming style. All the definitions, algorithms, lemmas and theorems have been described in the Coq proof assistant [Coq07]; the commented source script is the text of this report. While the proofs are not presented, the main hints are given. A consequence of the method is that there is no *a priori* doubt about the mathematical rigour of the presented results.

Samuel Eilenberg introduced in the chapter 10 of his book [Eil74], published in 1974, a notion of *Machine* which he claimed to be a *very efficient tool* for studying formal languages of the Chomsky hierarchy. They are sometimes referred to as *X-machines*. Many variants have appeared in the last twenty years [X-m] in several scientific domains different from formal languages.

Eilenberg machines define a general computational model. Assumed given an abstract data set X (it motivates X-machine terminology), a *machine* is defined as an automaton labelled with binary relations on X. Two generalizations result from this. Firstly, the set X abstracts the traditional tape, counters, stacks, *etc*, used by automata on words, push-down automata, transducers *etc*. Secondly, compared to functions, binary relations give a built-in notion of non-determinism. Many translations of others machines into Eilenberg machines are given in [Eil74]: automata, transducers, real-time transducers, two-way automata, push-down automata and Turing machines.

The remainder of this report gives the specification of Eilenberg machines with some restrictions. The restrictions are justified by effectiveness considerations concerning their simulation on computers [Raz08]. The original model of Eilenberg was defined in usual set theory which does not necessarily deal with computable objects. Using the Coq proof assistant [Coq07], we provide a formal specification of algorithms which simulate Eilenberg machines. Such algorithms define a reactive process in the spirit of the *reactive engine* introduced by Gérard Huet [Hue05]. Formal proofs are given ensuring the simulation is correct. The extraction code technology of Coq compile this formal development into an executable Ocaml [LDGV06] program that simulates any such Eilenberg machine.

2 The Specification

2.1 Required libraries

We require a few libraries from the Coq standard libraries.

```
Require Import List.
Require Import Bool.
Require Import Wf.
```

The library *Wf* is for well-founded relations. The accessibility predicate *Acc* will be very useful together with the associated well-founded induction principle *well_founded_ind*.

2.2 Streams for computable relations

We recall that ***unit*** is the singleton ML datatype containing the unique value denoted *tt*. In our specification streams are finitely defined objects for enumerating on demand (lazy lists).

```
Inductive stream (data: Set) : Set :=
  | EOS : stream data
  | Stream : data → (unit → stream data) → stream data.

Definition delay (data: Set) : Set := unit → stream data.
```

A stream value is either the empty stream *EOS* ("*End of Stream*") for encoding the empty enumeration or else a value *Stream d del* that provides the first element *d* of the enumeration and a value *del* as a delayed computation of the rest of the enumeration. Since this specification will be translated into ML and because ML computes with the restriction of λ-calculus to weak reduction, the computation of a value of type *delay data* such as *del* is delayed because it is a functional value. This well known technique permits computation on demand. Note that this technique would not apply in a programming language evaluating inside a function body (strong reduction in λ-calculus terminology). Remark that a stream value is necessarily finite because it is an ***Inductive*** definition. More general datatypes allowing potentially infinite values in Coq are provided by *CoInductive* construction.

Mathematical relations are objects that specify possibly non-deterministic computation. We restrict our study to endo-relations on a domain *data* which are defined as Coq functions from *data* to streams of *data*:

```
Definition relation (data: Set) : Set := data → stream data.
```

The choice of breaking the symmetry of relations is justified by the isomorphism between subsets of pairs of values of a set X and functions from X to subsets of X. The stream datatype may encode finite sets, thus relations of type *relation* have the following restriction: they associate necessarily a finite number of elements (a stream) to anyone. Remark that redundancies are allowed in stream enumerations.

Let us introduce a membership predicate for streams as the *In* predicate of the *List* library:

```
Inductive In_stream (data: Set) : data → stream data → Prop :=
  | InStr1 : ∀ (d: data) (del: delay data),
      In_stream data d (Stream data d del)
  | InStr2 : ∀ (d: data) (d': data) (del: delay data),
      In_delay data d del →
      In_stream data d (Stream data d' del)
```

```
with In_delay (data: Set): data → delay data → Prop :=
  | InDel: ∀ (d: data) (del: delay data),
      In_stream data d (del tt) → In_delay data d del.
```

It is an ***Inductive*** mutually recursive predicate on both *stream* and *delay* types.

2.3 A specific well-founded relation

We define a module type declaration for any well-founded set D over a relation R using the accessibility predicate *Acc* provided by the library *Wf*. Such objects are encapsulated in the followig module *WFMODULE*:

```
Module Type WFMODULE.
  Parameter D: Set.
  Parameter R: D → D → Prop.
  Hypothesis WFD: ∀ d:D , Acc R d.
End WFMODULE.
```

Let us now define a new well-founded relation *Rext* on lists of D values. It extends a *WFMODULE* and the extension is called *BiListExtension*.

```
Module BiListExtension (M: WFMODULE).
Import M.

Inductive Rext: list D → list D → Prop :=
  | Rext1 : ∀ (d:D) (l:list D), Rext l (d :: l)
  | Rext2 : ∀ (d1 d2 : D) (l : list D),
      R d1 d2 → Rext (d1 :: l) (d2 :: l)
  | Rext3 : ∀ (d1 d2 d3: D) (l : list D),
      R d1 d3 → R d2 d3 → Rext (d1 :: (d2 :: l)) (d3 :: l).
```

The relation *Rext* is a simple case of the multiset ordering extension [DM79] for the two following reasons:

1. The replacement of one element is performed only on the element at the head of the list and not at any position.

2. It replaces an element with at most two elements strictly less with respect to the relation R. The multiset ordering allows instead the replacement of one element by a finite multiset of others strictly less.

We prove that the *Rext* relation is well-founded if D is well-founded by relation R. The theorem is called *WfRext*:

```
Theorem WfRext : ∀ (l: list D), Acc Rext l.
```

The proof is by structural induction on the list l, using the fact that D is well-founded.

```
End BiListExtension.
```

2.4 Eilenberg machines

We define our subclass of Eilenberg machine first using relations specified as being of type *relation* then as a module containing the five parameters. Let us call such a module a *Machine*:

```
Module Type Machine.
  Parameter data: Set.

  Parameter state: Set.
  Parameter transition: state → list ((relation data) * state).
  Parameter initial: list state.
  Parameter terminal: state → bool.
End Machine.
```

The parameter *data* corresponds to an abstract set refered as X in the original work of Eilenberg. The other parameters *state*, *transition*, *initial* and *terminal* encode the automata structure traditionally written (Q, δ, I, T).

In the remainder of the report we will use the following notations:

- *d d' d1* for *data*,
- *s s' s1* for *state*,
- *ch ch' ch1* for **list** *((relation data) * state)*, shortcut of "choice",
- *rel rel' rel1* for *relation data*.

2.5 Formalization

Now we declare a module *Engine* that shall provide a process *characteristic_relation* that simulates an Eilenberg machine *M* of type *Machine*.

```
Module Engine ( M : Machine ).
Import M.
```

The following *cell* type is the "state" notion for the machine, it is the cartesian product of *data* and *state* of the automaton.

```
Definition cell : Set := prod data state.
```

The following *edge* property describes what is a correct reduction step in the machine *M*:

```
Definition edge d s rel d' s' : Prop :=
  In (rel,s') (transition s) /\ In_stream data d' (rel d).
```

It specifies a relation between two cells *(d,s)* and *(d',s')* linked by a relation *rel* from *M*.

A finite sequence of reduction steps is encoded in the manner of lists:

```
Inductive sequence : Set :=
  | Seq1: data → state → sequence
  | Seq2: data → state → (relation data) → sequence →
          sequence.
```

A sequence of reductions is not allowed to be empty, it contains at least one cell. This definition of sequence does not specify the fact that reductions are correct edges, this is specified using the *path* predicate defined below. For this purpose functions *hd_seq* and *tl_seq* give respectively the heading cell and the cell of the tail of a sequence *seq*:

```
Definition hd_seq (seq: sequence) : cell :=
  match seq with
  | Seq1 d s ⇒ (d,s)
  | Seq2 d s _ _ ⇒ (d,s)
  end.

Fixpoint tl_seq (seq: sequence) {struct seq} : cell :=
  match seq with
  | Seq1 d s ⇒ (d,s)
  | Seq2 _ _ _ seq' ⇒ tl_seq seq'
  end.
```

A correct sequence is then defined as the following inductive predicate:

```
Inductive path: sequence → Prop :=
  | Path1: ∀ d s, path (Seq1 d s)
  | Path2: ∀ d s rel d' s' seq,
    path seq → (d',s') = hd_seq seq → edge d s rel d' s' →
    path (Seq2 d s rel seq).
```

The following two functions *init* and *term* ensures that the state of a cell c are initial and terminal with respect to *initial* and *terminal* parameters of the machine M.

```
Definition init (c: cell): Prop:= In (snd c) initial.

Definition term (c: cell): Prop:= terminal (snd c) = true.
```

Finally we formalize a data d' to be a solution of data d with respect to machine M as the following:

```
Definition Solution (d:data) (d':data): Prop :=
  ∃ seq:sequence,
  path seq /\ d = fst (hd_seq seq) /\ d' = fst (tl_seq seq) /\
  init (hd_seq seq) /\ term (tl_seq seq).
```

2.6 A machine with nœtherian reductions

Our restriction of Eilenberg machine consists in two restrictions:

1. the above specification of relation operation as function from data to finite stream of data,
2. a nœtherian condition on cells along a path. That is to forbid infinite path.

For this second purpose we introduce a relation on cells that we assume to be well-founded (nœtherian):

```
Definition Rcell (c' c: cell) : Prop := ∃ rel:relation data,
  edge (fst c) (snd c) rel (fst c') (snd c').

Hypothesis WfRcell : ∀ c:cell, Acc Rcell c.
```

At this point we have all we need to enunciate the theorem we aim at proving:

```
Theorem goal : ∃ f : (relation data),
  ∀ (d d': data),
  Solution d d' ⟷ In_stream data d' (f d).
```

It says that there exists a functional relation f simulating correctly the machine M with respect to the *Solution* specification. In the remainder we will provide such a f which is a generalization of the reactive engine of Gérard Huet and prove the theorem for this function. Even if our function f will be a reactive process by nature we will show that f terminates for any data d, this allows us to define it using the ***Fixpoint*** construction of Coq. We will say that f is the characteristic relation of M and thus call it *characteristic_relation* .

2.7 Resumptions for non-deterministic search

Let us introduce an abbreviation for the list of transitions of the machine M

```
Definition choice: Set := list ((relation data) * state).
```

We define the following two datatypes manipulated by the reactive engine:

```
Inductive backtrack: Set :=
  | Advance : data → state → backtrack
  | Choose : data → state → choice → (relation data) →
             (delay data) → state → backtrack.

Definition resumption: Set := list backtrack.
```

Eilenberg machines are possibly non-deterministic and need thus a backtracking mechanism for their simulation. Values of type *backtrack* allow to save the multiple choices due to the non-deterministic nature of the machine. The reactive engine will stack such backtrack values in a resumption of type *resumption*.

The predicate *WellFormedBack* specifies a well-formedness invariant of backtrack values:

```
Inductive WellFormedBack: backtrack → Prop :=
  | WFB1: ∀ d s, WellFormedBack (Advance d s)
  | WFB2: ∀ d s ch rel del s',
    In (rel,s') (transition s) →
    (∀ d1, In_stream data d1 (del tt) →
    In_stream data d1 (rel d)) →
    incl ch (transition s) →
    WellFormedBack (Choose d s ch rel del s').
```

Well formed resumptions are lists of well formed backtrack values:

```
Definition WellFormedRes (res:resumption): Prop :=
  ∀ b:backtrack, In b res → WellFormedBack b.
```

We state some useful lemmas concerning backtrack and resumption values:

```
Lemma lem_incl: ∀ ch (rel:relation data) (s':state) rest s,
  (ch = ((rel, s') :: rest)) →
  incl ch (transition s) →
  incl rest (transition s).

Lemma WellFormedRes_cons : ∀ b res,
  WellFormedBack b → WellFormedRes res → WellFormedRes (b :: res).

Lemma lemma2 : ∀ d s ch res rel s' rest d' del,
  (ch = ((rel, s') :: rest)) →
  incl ch (transition s) →
  rel d = Stream data d' del →
  WellFormedRes res →
  WellFormedRes ((Choose d s rest rel del s') :: res).

Lemma WellFormedRes_destr: ∀ res b res',
  res = (b :: res') →
  WellFormedRes res →
  WellFormedRes res'.

Lemma lemma3 : ∀ res b res' d s ch rel del s',
  WellFormedRes res →
  res = (b :: res') →
  b = Choose d s ch rel del s' →
  incl ch (transition s).

Lemma lemma4: ∀ res b res' d s ch rel del s' d' del',
  WellFormedRes res →
  res = (b :: res') →
  b = Choose d s ch rel del s' →
  del tt = Stream data d' del' →
  WellFormedRes ((Choose d s ch rel del' s') :: res').
```

2.8 Constructing the termination proof

In Coq we need to prove the termination of functions declared using ***Fixpoint***. Let us now construct our argument of termination. First we introduce a measure *chi* which is a triple consisting of a cell and two natural numbers:

```
Inductive chi : Set := | Chi: cell → nat → nat → chi.
```

The two natural numbers are respectively the length of a choice list and the length of a stream. The first one is already available in the library *List* with the function *length* and the second one is the function *length_del* defined here as the following:

```
Fixpoint length_str (str: stream data) {struct str} : nat :=
  match str with
  | EOS ⇒ O
```

```
  | Stream _ del ⇒ S (length_str (del tt))
  end.

Definition length_del (del: delay data) : nat :=
  length_str (del tt).
```

A *chi* measure is associated to backtrack values as the following:

```
Definition chi_back (back:backtrack) : chi :=
  match back with
  | Advance d s ⇒
    Chi (d, s) (S (S (length (transition s)))) O
  | Choose d s ch rel del s' ⇒
    Chi (d, s) (length ch) (S (length_del del))
  end.
```

The final measure value we will consider for proving the termination is list of *chi* measure. Such values are associated to any resumption using the *map* function of module *List*:

```
Definition chi_res (res:resumption) : list chi :=
  map chi_back res.
```

Now we are to introduce a well-founded relation on lists of *chi* and use it for the termination of the reactive engine.

The *Rnat* relation is simply the predecessor relation on natural numbers:

```
Definition Rnat (n' n : nat) : Prop := n = S n'.
```

Rnat is well-founded:

```
Lemma WfRnat: ∀ n:nat, Acc Rnat n.
```

The proof is by structural induction on *n*.

Let *RChi* be a relation on *chi* which is a specific lexicographic ordering.

```
Inductive RChi : chi → chi → Prop :=
  | RC1 : ∀ c' n1' n2' c n1 n2,
    Rcell c' c → RChi (Chi c' n1' n2') (Chi c n1 n2)
  | RC2: ∀ c n1' n2' n1 n2,
    Rnat n1' n1 →  RChi (Chi c n1' n2') (Chi c n1 n2)
  | RC3: ∀ c (n1:nat) n2' n2,
    Rnat n2' n2 →  RChi (Chi c n1 n2') (Chi c n1 n2).
```

The relation *RChi* is also well-founded.

```
Lemma WfRChi: ∀ v:chi, Acc RChi v.
```

Now we extend *RChi* as a *BiListExtension* in the module called *MyUtil*:

```
Module ModuleWfChiRes.
  Definition D := chi.
  Definition R := RChi.
  Definition WFD := WfRChi.
End ModuleWfChiRes.
```

```
Module MyUtil := BiListExtension(ModuleWfChiRes).
Import MyUtil.
```

We thus obtain the corresponding instance of the well-founded *Rext* relation on lists of *chi*.

2.9 The reactive engine definition

We are now going to introduce the so-called reactive engine. It aims at simulating an Eilenberg machine which is a priori a non-deterministic machine. We make a distinction between the terminology "engine" and "machine" since our engine is a machine which is **deterministic**.

The seven following lemmas *dom_react_inv1*, *dom_choose_inv1*, *dom_choose_inv2*, *dom_choose_inv3*, *dom_continue_inv1*, *dom_continue_inv2* and *dom_continue_inv3* are needed for the termination of the reactive process:

```
Lemma dom_react_inv1 : ∀ d s res,
  Acc Rext ((Chi (d, s) (S (length (transition s))) O) :: (chi_res res)) →
  Acc Rext ((Chi (d, s) (length (transition s)) O) :: (chi_res res)).

Lemma dom_choose_inv1 : ∀ d s (ch:choice) res,
  Acc Rext ((Chi (d, s) (length ch) O) :: (chi_res res)) →
  Acc Rext (chi_res res).

Lemma dom_choose_inv2 : ∀ d s (ch:choice) res rel s' rest,
  Acc Rext ((Chi (d, s) (length ch) O) :: (chi_res res)) →
  ch = (rel, s') :: rest →
  Acc Rext ((Chi (d, s) (length rest) O) :: (chi_res res)).

Lemma dom_choose_inv3 : ∀ d s ch res rel s' rest d' del,
  Acc Rext ((Chi (d, s) (length ch) O) :: (chi_res res)) →
  ch = (rel, s') :: rest → rel d = Stream data d' del →
  incl ch (transition s) →
  Acc Rext ((Chi (d', s') (S (length (transition s'))) O) ::
            (chi_res ((Choose d s rest rel del s') :: res))).

Lemma dom_continue_inv1 : ∀ res back res' d s rest rel del s',
  Acc Rext (chi_res res) →
  res = back :: res' →
  back = Choose d s rest rel del s' →
  del tt = EOS data →
  Acc Rext ((Chi (d, s) (length rest) 0) :: (chi_res res')).

Lemma dom_continue_inv2 :
  ∀ res back res' d s rest rel del s' d' del',
  Acc Rext (chi_res res) →
  res = back :: res' →
  back = Choose d s rest rel del s' →
  del tt = Stream data d' del' →
```

```
  WellFormedRes res →
  Acc Rext ((Chi (d', s') (S (length (transition s'))) 0) ::
            (chi_res ((Choose d s rest rel del' s') :: res'))).

Lemma dom_continue_inv3 : ∀ res back res' d s,
  Acc Rext (chi_res res) →
  res = back :: res' →
  back = Advance d s →
  Acc Rext ((Chi (d, s) (S (length (transition s))) 0) ::
            (chi_res res')).
```

The central part of the reactive engine is defined as three mutually recursive functions *react*, *choose* and *continue*.

```
Fixpoint react (d : data) (s : state) (res : resumption)
  (h1: WellFormedRes res)
  (h : Acc Rext ((Chi (d, s) (S (length (transition s))) O) :: (chi_res res)))
  {struct h} : stream data :=
  if terminal s
  then Stream data d (fun x:unit ⇒ choose d s (transition s) res h1
    (incl_refl (transition s)) (dom_react_inv1 d s res h))
  else choose d s (transition s) res
         h1 (incl_refl (transition s)) (dom_react_inv1 d s res h)

with choose (d : data) (s:state) (ch : choice) (res : resumption)
  (h1: WellFormedRes res)
  (h2 : incl ch (transition s))
  (h : Acc Rext ((Chi (d, s) (length ch) O) :: (chi_res res)))
  {struct h} : stream data :=
  match ch as e1 return ch = e1 → (stream data) with
  | nil ⇒ fun eq1 ⇒ continue res h1 (dom_choose_inv1 d s ch res h)
  | ((rel, s') :: rest) ⇒ fun eq1 ⇒
    match (rel d) as str1 return (rel d) = str1 → (stream data) with
    | EOS ⇒ fun eq2 ⇒
      choose d s rest res
        h1 (lem_incl ch rel s' rest s eq1 h2)
        (dom_choose_inv2 d s ch res rel s' rest h eq1)
    | Stream d' del ⇒ fun eq2 ⇒
      react d' s' ((Choose d s rest rel del s') :: res)
        (lemma2 d s ch res rel s' rest d' del eq1 h2 eq2 h1)
        (dom_choose_inv3 d s ch res rel s' rest d' del h eq1 eq2 h2)
    end (refl_equal (rel d))
  end (refl_equal ch)

with continue (res : resumption)
  (h1: WellFormedRes res)
  (h : Acc Rext (chi_res res))
  {struct h} : stream data :=
  match res as res1 return res = res1 → (stream data) with
  | nil ⇒ fun eq1 ⇒ EOS data
  | (back :: res') ⇒ fun eq1 ⇒
```

```
    match back as back1 return back = back1 → (stream data) with
    | Advance d s ⇒ fun eq2 ⇒
      react d s res'
        (WellFormedRes_destr res back res' eq1 h1)
        (dom_continue_inv3 res back res' d s h eq1 eq2)
    | Choose d s rest rel del s' ⇒ fun eq2 ⇒
      match del tt as str1 return del tt = str1 → (stream data) with
      | EOS ⇒ fun eq3 ⇒
        choose d s rest res'
          (WellFormedRes_destr res back res' eq1 h1)
          (lemma3 res back res' d s rest rel del s' h1 eq1 eq2)
          (dom_continue_inv1 res back res' d s rest rel del s' h eq1 eq2 eq3)
      | Stream d' del' ⇒ fun eq3 ⇒
        react d' s' ((Choose d s rest rel del' s') :: res')
          (lemma4 res back res' d s rest rel del s' d' del' h1 eq1 eq2 eq3)
          (dom_continue_inv2 res back res' d s rest rel del s'
                             d' del' h eq1 eq2 eq3 h1)
      end (refl_equal (del tt))
    end (refl_equal back)
  end (refl_equal res).
```

First omit parameters *h1*, *h2* and *h* in this definition. The function *react* checks whether the state is terminal and then provides an element of the stream delaying the rest of the exploration calling to the function *choose*. This function *choose* performs the non-deterministic search over transitions, choosing them in the natural order induced by the ***list*** data structure. The function *continue* manages the backtracking mechanism and the enumeration of finite streams of relations, it always chooses to backtrack on the last pushed value in the resumption. Remark that these three mutually recursive functions do not use any side effect and are written in a pure functional style completely tail-recursive using the resumption as a continuation mechanism.

The three functions ensure that arguments computed are well formed as a post-condition if arguments are well-formed as a pre-condition in the predicate *h1*. The predicate *h2* ensures a part of the well-formedness of the list of transitions in function *choose*. The termination is ensured using the accessibility predicate on list of *chi* in the predicate *h*.

Let *d* be a data, a machine may be initialized with any of its initial states. This non-deterministism may be encoded as a resumption containing only *Advance* constructors. This is performed by the following *init_res* function:

```
Fixpoint init_res (d:data) (l: list state) (acc: resumption)
  {struct l} : resumption :=
  match l with
  | nil ⇒ acc
  | (s :: rest) ⇒ init_res d rest ((Advance d s) :: acc)
  end.
```

The parameter *acc* is an accumulator for the resulting resumption. A call to *init_res* would be an empty accumulator and the parameter *l* equal to *initial*

(the list of initial states fo the machine). A resumption computed by *init_res* is easily proved to be well formed:

```
Lemma lemma5 : ∀ d l , WellFormedRes (init_res d l nil).
```

Finally we define the function *characteristic_relation* of expected type *relation data* that, should have the following functionality: given a data *d*, the stream *characteristic_relation d* contains exactly all data *d'* such that the predicate *Solution d d'* is true.

```
Definition characteristic_relation : relation data :=
  fun (d:data) ⇒ continue (init_res d initial nil)
    (lemma5 d initial) (WfRext (chi_res (init_res d initial nil))).
```

The function *characteristic_relation* is the function *f* of the theorem *goal* we were looking for.

2.10 Correction of the reactive engine: soundness and completeness

We are to prove the soundness and the completeness of the function *characteristic_relation* . For this purpose we need to prove soundness and completeness of its central part, that is *react*, *choose* and *continue* functions. The following predicates specifies invariants of those functions:

```
Definition PartSol (d:data) (s:state) (d':data): Prop :=
  ∃ seq:sequence,
  ( path seq /\ (d,s) = hd_seq seq /\
    d' = fst (tl_seq seq) /\ term (tl_seq seq)
  ).
```

The porperty *PartSol d s d'* holds if and only if the cell *(d,s)* leads to a terminal cell with data *d'*. We extend this predicate on backtrack and resumption values:

```
Inductive PartSolBack: backtrack → data → Prop:=
  | SB1: ∀ d s d',
    PartSol d s d' → PartSolBack (Advance d s) d'
  | SB2: ∀ d s ch a del s1 d',
    PartSol d s d' → PartSolBack (Choose d s ch a del s1) d'.

Definition PartSolRes (res: resumption) (d': data): Prop:=
  ∃ b:backtrack, In b res /\ PartSolBack b d'.
```

The following predicate is a variant of *PartSol* useful for the specification of the *choose* function.

```
Definition PartSol_choice (d:data) (ch : choice) (d':data): Prop :=
  ∃ rel, ∃ s1, ∃ d1,
  In (rel, s1) ch /\ (In_stream data d1 (rel d)) /\ PartSol d1 s1 d'.
```

We extend it to backtrack and resumption values:

```
Inductive PartSolBack2: backtrack → data → Prop:=
  | SB3: ∀ d s d', PartSol d s d' → PartSolBack2 (Advance d s) d'
  | SB4: ∀ d s ch rel del s1 d',
```

```
    PartSol_choice d ch d' →
    PartSolBack2 (Choose d s ch rel del s1) d'
  | SB5: ∀ d s ch rel del s1 d1 d',
    (In_delay data d1 del /\ PartSol d1 s1 d') →
    PartSolBack2 (Choose d s ch rel del s1) d'.

Definition PartSolRes2 (res: resumption) (d': data): Prop:=
  ∃ b:backtrack, In b res /\ PartSolBack2 b d'.
```

The soundness lemma of the reactive process concerns the three functions *react*, *choose* and *continue* and may be stated as follows:

```
Lemma soundness_lemma: ∀ (v: list chi),
  ((∀ d s res h1 h d',
    v = (Chi (d, s) (S (length (transition s))) 0 :: chi_res res) →
    In_stream data d' (react d s res h1 h) →
    PartSol d s d' \/ PartSolRes res d' )
  /\(∀ d s ch res h1 h2 h d',
    v = (Chi (d, s) (length ch) 0 :: chi_res res) →
    In_stream data d' (choose d s ch res h1 h2 h) →
    PartSol d s d' \/ PartSolRes res d')
  /\(∀ res h1 h d',
    v = chi_res res →
    In_stream data d' (continue res h1 h) →
    PartSolRes res d')
  ).
```

The proof is a routine case analysis by simultaneous well-founded induction over the measure *v*.

Using this soundness lemma we prove the soundness theorem for the reactive engine:

```
Theorem soundness: ∀ (d d': data),
  In_stream data d' (characteristic_relation d) → Solution d d'.
```

In the same way we give the completeness lemma which is a bit stronger than the converse of the soundness lemma since it uses *PartSolRes2* and *ParSol_choice* instead of *PartSolRes* and *PartSol*.

```
Lemma completeness_lemma: ∀ (v: list chi),
  ((∀ d s res h1 h d',
    v = (Chi (d,s) (S (length (transition s))) 0 :: chi_res res) →
    PartSol d s d' \/ PartSolRes2 res d' →
    In_stream data d' (react d s res h1 h))
  /\(∀ d s ch res h1 h2 h d',
    v = (Chi (d,s) (length ch) 0 :: chi_res res) →
    PartSol_choice d ch d' \/ PartSolRes2 res d' →
    In_stream data d' (choose d s ch res h1 h2 h))
  /\(∀ res h1 h d',
    v = chi_res res →
    PartSolRes2 res d' →
    In_stream data d' (continue res h1 h))
  ).
```

The proof is again a routine case analysis by simultaneous well-founded induction over the measure v.

Using the completeness lemma we prove the completeness of the reactive engine:

```
Theorem completeness: ∀ (d d': data),
  Solution d d' → In_stream data d' (characteristic_relation d).
```

The correction of the reactive engine combines both soundness and completeness:

```
Theorem correction : ∀ (d d': data),
  Solution d d' ⟷ In_stream data d' (characteristic_relation d).
```

```
End Engine.
```

This work represents 390 lines of Coq specification and 1150 lines of proof scripts.

3 Code extraction

The formal development above used as specification language the *Calculus of Inductive Constructions*, a version of higher-order logic suited for abstract mathematical development, but also for constructive reasoning about computational objects. Here the sort ***Prop*** is needed for logical properties, when the sort ***Set*** is used for computational objects. This allows a technique of code extraction which can be evoked for extracting an actual computer program verifying the logical specification. Thus, using *OCaml* as the target extraction language, the Coq proof assistant provides mechanically the following program:

```
type unit0 =
  | Tt

type bool =
  | True
  | False

type ('a, 'b) prod =
  | Pair of 'a * 'b

type 'a list =
  | Nil
  | Cons of 'a * 'a list

type 'data stream =
  | EOS
  | Stream of 'data * (unit0 → 'data stream)

type 'data delay = unit0 → 'data stream
```

```
type 'data relation = 'data → 'data stream

module type Kernel = sig
  type data
  type state
  val transition : state → (data relation, state) prod list
  val initial : state list
  val terminal : state → bool
end

module Engine = functor (M:Kernel) → struct

type choice = (M.data relation, M.state) prod list

type backtrack =
  | Advance of M.data * M.state
  | Choose of M.data * M.state * choice * M.data relation *
    M.data delay * M.state

type resumption = backtrack list

let rec react d s res =
  match M.terminal s with
  | True → Stream (d, (fun x → choose d s (M.transition s) res))
  | False → choose d s (M.transition s) res

and choose d s ch res =
  match ch with
  | Nil → continue res
  | Cons (p, rest) →
    let Pair (rel, s') = p in
    match rel d with
    | EOS → choose d s rest res
    | Stream (d', del) →
      react d' s' (Cons ((Choose (d, s, rest, rel, del, s')), res))

and continue = function
  | Nil → EOS
  | Cons (back, res') →
    match back with
    | Advance (d, s) → react d s res'
    | Choose (d, s, rest, rel, del, s') →
      match del Tt with
      | EOS → choose d s rest res'
      | Stream (d', del') →
        react d' s' (Cons ((Choose (d, s, rest, rel, del', s')), res'))

let rec init_res d l acc =
  match l with
  | Nil → acc
```

```
  | Cons (s, rest) → init_res d rest (Cons ((Advance (d, s)), acc))

let characteristic_relation d =
  continue (init_res d M.initial Nil)

end
```

Despite its high-level character, this program is computationally efficient. Note that all recursion calls are terminal, thus implemented by jumps. The extracted program could be expected from an OCaml programmer because it is not cluttered with logical justifications which are not needed for computational aspects: the predicates *h*, *h1* and *h2* appearing in the definitions of *react*, *choose* and *continue* are erased. The reader will check that this program is indeed very close to the original reactive engine introduced in [Hue05] or its extensions in [HR06].

4 Conclusion

In this report we present a specification of a restriction of the Eilenberg Machine model. The restriction consists in two parts which concern:

1. the specification of the relations labelling the machine,
2. the finiteness of the machine execution.

Those restrictions allow us a complete formalization of a reactive process simulating such machines using the Coq proof assistant. This reactive process, called the *reactive engine* from Gérard Huet work, is proved to terminate and to be correct.

Our methodology has been incremental:

1. Define the algorithm in a functional style.
2. Sketch an informal hand proof including well-formedness arguments.
3. Adapt the two previous points formally to the Coq proof assistant.
4. Extract automatically from this development an efficient high-level code more or less isomorphic to the original one.

Finally, we believe that the Eilenberg machines model should offer a base for the design of a high-level language to specify and solve problems in language theory as presented in [Raz08].

Acknowledgments

Gérard Huet took an essential part in the elaboration of this paper with stimulating discussions. Matthieu Sozeau helped a lot to finalize the Coq development.

References

[Coq07] The Coq proof assistant. Software and documentation available on the Web, `http://coq.inria.fr/`, 1995–2007.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[Eil74] Samuel Eilenberg. *Automata, languages, and machines, Volume A*. Academic Press, 1974.

[HR06] Gérard Huet and Benoît Razet. The reactive engine for modular transducers. In Jean-Pierre Jouannaud Kokichi Futatsugi and José Meseguer, editors, *Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 355–374. Springer-Verlag LNCS vol. 4060, 2006.

[Hue05] Gérard Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional programming*, 15, 2005.

[LDGV06] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2006.

[Raz08] Benoît Razet. Finite Eilenberg machines. Research Report 6486, INRIA, 02 2008.

[X-m] Theory of X-machines. `http://www.x-machines.com`.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399