Simulating Finite Eilenberg Machines with a Reactive Engine

Benoît Razet[1]

INRIA Paris-Rocquencourt

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 the subclass of *finite Eilenberg machines* for which we provide an executable complete simulator. This program is specified using the Coq proof assistant. The correctness of the algorithm is also proved formally and mechanically verified using Coq. Using its extraction mechanism, the Coq proof assistant allows to translate the specification into an executable OCaml program. The algorithm and specification are inspired from the reactive engine of Gérard Huet. The finite Eilenberg machines model includes deterministic and non-deterministic automata (DFA and NFA) but also real-time transducers. As an example, we present a pushdown automaton (PDA) recognizing ambiguous terms of the λ-calculus. We show that this pushdown automaton is a finite Eilenberg machine, then the simulation using the reactive engine provides a complete recognizer for this particular context-free language.

Keywords: Automata, Eilenberg machines, Coq

1 Introduction

Samuel Eilenberg introduced in the chapter 10 of his book *"Automata, Languages and Machines"* [4] a general computational model, now called *Eilenberg machines*, meant to study the formal languages of the Chomsky hierachy. In this formalism a *machine* is defined as an automaton labelled with binary relations over a set X. The set X abstracts data structures common in language theory such as tapes, counters, stacks, *etc*, used by automata on words, push-down automata, transducers *etc*. Moreover binary relations give a built-in notion of non-determinism. Many *translations* [4] of usual computational models such as automata, transducers, real-time transducers, two-way automata, push-down automata and Turing machines can be presented as Eilenberg machines.

The generality of this model is interesting for specifying problems that use many different kind of machines. Unfortunately the expressive power of this formalism and its presentation in set theory makes it unrealistic from an effective point of

[1] Email: benoit.razet@inria.fr

view. For these reasons we have introduced a restriction of the model called *finite Eilenberg machines* in a recent paper [8]. We have provided a simulation algorithm, in the spirit of the *reactive engine* of Gérard Huet [5]. It is written in OCaml [7] and we have given an informal proof of its correctness.

The proof of correctness uses an inductive principle based on the multiset ordering [3] over three mutually recursive predicates. Due to the subtlety of the termination argument we find necessary to formalize the simulation in a proof assistant. Using the Coq proof assistant [1] and its PROGRAM extension [10], we provide a specification of finite Eilenberg machines along with mechanized proof of its termination and correctness.

The remainder of this paper is organized as follows. Section 2 provides the specification of finite Eilenberg machines. Section 3 presents the termination proof techniques needed for the *reactive engine* definition provided in Section 4. Section 5 presents the proof of correctness of the reactive engine. Section 6 discusses the program obtained by the extraction mechanism of Coq. Section 7 provides an example of a pushdown automaton recognizing words of the λ-calculus as a finite Eilenberg machine and discusses the efficiency of the reactive engine which provides all solutions with respect to the pushdown automaton.

2 Finite Eilenberg machines

We recall that `unit` is the singleton 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 binary 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. A `stream` value may encode a finite set, thus relations of type `relation` are *locally* *finite relations* that is: for all data `d` the set of elements in relation with `d` is finite.

Let us introduce a membership predicate for streams similar to the predicate `In` in library `List`:

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

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

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

We define our subclass of Eilenberg machine first using relations specified as being of type `relation` then as a module containing 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.
```

Libraries `List` and `Bool` provide datastructure types `list` and `bool`. The parameter `data` corresponds to an abstract set referred as X in the original work of Eilenberg. The other parameters `state`, `transition`, `initial` and `terminal` encode the automaton structure traditionally written (Q, δ, I, T). A machine is an automaton labelled with relations instead of a traditional alphabet. In the remainder we will use the following notations: `d d' d1` for a `data` value, `s s' s1` for a `state` value, `ch ch' ch1` for a `list ((relation data) * state)` value and `rel rel' rel1` for values of type `relation data`.

Now we assume given a machine `M` of type `Machine` declaring a functor with `M` as argument.

```
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`.

```
Definition cell : Set := (data * state).
```

The following `edge` property specifies 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).
```

Intuitively, it is a relation between two cells `(d,s)` and `(d',s')` linked by a relation `rel`; `(rel,s')` is a transition of state `s` and `d d'` are in relation by `rel`.

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* ensure that the state of a cell *c* is 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 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).
```

The data *d'* is a solution of *d* if and only if there exists a path between them beginning with an initial state and ending with a terminal state.

We recall that the *finite Eilenberg machine* model [8] consists in two restrictions with respect to Eilenberg machines:

(i) The above specification of relation operation as function from data to finite stream of data. *(first condition)*

(ii) "Computations" are necessarily finite. *(second condition)*

For this second condition we introduce a relation on cells and we assume it is a well-founded relation:

```
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.
```

It will be clear in Section 3 why this hypothesis corresponds to the non existence of infinite path. 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`.

3 Construction of the termination argument for the reactive engine

In this section we give termination proof techniques for proving the termination of the process that simulates any finite Eilenberg machine. The techniques rely on the use of well-founded relations. Coq provides a library for this purpose called `Wf`. Let us recall basic notions of the library. First, a binary relation on elements of type `A` is a predicate `R : A →A → `***Prop***. A well-founded relation is a relation for which the chains of elements *left*-related by `R` are necessarily finite. This is formalized by the following *accessibility* predicate `Acc`:

```
Inductive Acc (R : A → A → Prop) (x: A) : Prop :=
  | Acc_intro : (∀ y:A, R y x → Acc R y) → Acc R x.
```

Intuitively, if `Acc R x` holds then there cannot be an infinite sequence $\mathtt{x}_i (i \in \mathbb{N})$ such that R $\mathtt{x}_{i+1}$ $\mathtt{x}_i$ holds for any index i. A relation `R` is well-founded if every element is accessible:

```
Definition well_founded (R : A → A → Prop) := ∀ a:A, Acc R a.
```

We also consider the well-founded induction principle `well_founded_ind`:

```
Theorem well_founded_ind : ∀ (R : A → A → Prop),
  well_founded R →
    ∀ P : A → Prop,
      (∀ x : A, (∀ y : A, R y x → P y) → P x) → ∀ a : A, P a.
```

Further explanations are provided in the book of Yves Bertot and Pierre Castéran [2]. Now we shall define a type of module containing a set `D` and a well-founded relation `R` and call this module type `WFMODULE`:

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

The traditional well-founded relation on lists is the ordering on their length. Let us define a new well-founded relation on lists called `BiListExtension`:

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

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 [3] for the two following reasons:

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

(ii) 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 `Rext` is well-founded:

```
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.
```

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

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

We define the following two datatypes used 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.
```

Finite 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`. A value `Advance d s` means being on cell `(d,s)`. A value `Backtrack d s ch rel del s'` means being on cell `(d,s)`, looking at transition `(rel,s')` of `(transition s)`, `del` being a delay of `(rel d)` and `ch` transitions included in `transition s`. This is specified by the following `WellFormedBack` predicate which is an invariant of any backtrack value constructed:

```
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.
```

In Coq we need to prove the termination of recursive functions. 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 ⇒ 0
  | 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) (2 + (length (transition s))) 0
  | 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 *Pred* relation is simply the predecessor relation on natural numbers which is well-founded:

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

Lemma WfPred: ∀ n:nat, Acc Pred 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,
    Pred n1' n1 → RChi (Chi c n1' n2') (Chi c n1 n2)
  | RC3: ∀ c n1 n2' n2,
    Pred 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*.

4 The reactive engine

We are now going to introduce the so-called *reactive engine* which simulates the finite Eilenberg machine *M* which is *a priori* a non-deterministic machine; a data *d* may have many solutions *d'* with respect to the *Solution* predicate.

The central part of the reactive engine is defined as three mutually recursive functions *react*, *choose* and *continue* with the PROGRAM Coq extension [10].

```
Program Fixpoint react (d : data) (s : state) (res : resumption)
  (h1 : WellFormedRes res)
  (h : Acc Rext ((Chi (d, s) (S (length (transition s))) 0) :: (chi_res res)))
  {struct h} : (stream data) :=
  if terminal s
  then Stream data d (fun x:unit ⇒ choose d s (transition s) res h1 _ _)
  else choose d s (transition s) res h1 _ _

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) 0) :: (chi_res res)))
  {struct h} : (stream data) :=
  match ch with
  | [] ⇒ continue res h1 _
  | (rel, s') :: rest ⇒
    match (rel d) with
    | EOS ⇒ choose d s rest res h1 _ _
    | Stream d' del ⇒
      react d' s' ((Choose d s rest rel del s') :: res) _ _
    end
  end

with continue (res : resumption)
  (h1 : WellFormedRes res)
  (h : Acc Rext (chi_res res)) {struct h} : (stream data) :=
  match res with
  | [] ⇒ EOS data
  | 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' ((Choose d s rest rel del' s') :: res') _ _
      end
    end
  end.
```

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 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`; the property `WfRext` is used to ensure that all recursive calls are performed with structurally less argument of `h`.

Using the PROGRAM extension of Coq [10] we obtain a readable program definition. It is due to two features of PROGRAM. First, it is allowed to give function definitions without providing all logical justifications which are delayed as proof obligations. Each underscore character in the body of the function creates a proof obligation according to function declarations. Secondly, PROGRAM brings the following enhancement concerning the *dependent pattern matching*:

```
match v return T with
| v1  ⇒ t1
| ... ⇒ ...
| vn  ⇒ tn
end
```

is replaced with

```
match v as x return v = x → T with
| v1  ⇒ (fun eq ⇒ t1)
| ... ⇒ ...
| vn  ⇒ (fun eq ⇒ tn)
end (refl_equal v)
```

One shall appreciate the improvements brought by PROGRAM in the definition of the reactive engine compared with the same reactive engine without PROGRAM as presented in the research report [9]. The proof obligations generated by PROGRAM are the same predicates as the ones explicitly provided in the reactive engine definition in the research report; which are the properties due to program invariants corresponding to *h1 h2* and the termination argument corresponding to *h*.

Let *d* be a data, a machine may be initialized with any of its initial states. We encode this non-deterministism 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
  | [] ⇒ acc
  | (s :: rest) ⇒ init_res d rest ((Advance d s) :: acc)
  end.
```

The parameter *acc* is an accumulator for the resulting resumption. The function *init_res* is initialized with an empty accumulator and the parameter *l* equal to *initial* (the list of initial states of the machine). A resumption computed by *init_res* is easily proved to be well formed:

```
Lemma lemma_init: ∀ d l, WellFormedRes (init_res d l []).
```

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 [])
    (lemma_init d initial) (WfRext (chi_res (init_res d initial []))).
```

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

Remark. Engine *versus* Machine. We make a distinction between the terminology “engine” and “machine”. A machine can be non-deterministic whereas an engine is a deterministic process able to simulate a non-deterministic one. Finite Eilenberg machines describe non-deterministic computations which are enumerated by a deterministic process: the reactive engine.

5 Correctness 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 the three mutually recursive functions *react*, *choose* and *continue* functions. The following predicates specify 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 property *PartSol d s d'* holds if and only if the cell *(d,s)* begins a path with 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 enriches `PartSol` for the specification of `choice` transitions.

```
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 three functions `react`, `choose` and `continue` is 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 case analysis by simultaneous well-founded induction over the measure *v*.

Using this soundness lemma we prove the soundness theorem for the reactive engine: For all data `d` and `d'`, if `d'` is enumerated by the reactive engine applied to `d` then it `d'` is a solution of `d`.

```
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 case analysis by simultaneous well-founded induction over the measure `v`.

Using the completeness lemma we prove the completeness of the reactive engine: For all data `d` and `d'`, if `d'` is a solution of `d` then `d'` is enumerated by the reactive engine applied to `d`.

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

The correctness of the reactive engine combines both soundness and completeness: The reactive engine enumerates exactly all solutions:

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

End Engine.
```

6 Program 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 program 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 'a list =
  | Nil
  | Cons of 'a * 'a list

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

type 'data delay = unit → 'data stream

type 'data relation = 'data → 'data stream

module type Machine = sig
  type data
  type state
  val transition : state → (data relation * state) list
  val initial : state list
  val terminal : state → bool
end

module Engine = functor (M:Machine) → struct

type choice = (M.data relation * M.state) 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 =
  if M.terminal s
  then Stream (d, (fun x → choose d s (M.transition s) res))
  else choose d s (M.transition s) res

and choose d s ch res =
  match ch with
```

```
    | Nil → continue res
    | Cons (p, rest) →
      let (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 () 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 one of *finite Eilenberg machines* [8] but also to the original reactive engine introduced by Gérard Huet [5] or even to its extensions [6].

7 Example : a pushdown automaton recognizing words of the λ-calculus

We discuss in this section the efficiency of the reactive engine obtained above. For this purpose let us embed into the finite Eilenberg machine model a pushdown automaton recognizing terms of the λ-calculus. Then we will discuss the efficiency of the reactive engine performing the search of solutions upon this pushdown automaton.

Consider the following ambiguous grammar for λ-terms:

$$\begin{array}{ccll} T & := & \mathtt{x} & \text{(variable)} \\ & | & \lambda\mathtt{x}{\cdot}T & \text{(abstraction)} \\ & | & T\mathtt{@}T & \text{(application)} \\ & | & (T) & \end{array}$$

Terminal symbols of this language are x, λ, ·, @, (,) and @. The symbol T is *non-terminal*. Following this grammar the λ-term "λx.x@λx.x" may be recognized as "λx.(x@λx.x)" but also as "(λx.x)@(λx.x)". Thus a complete parsing algorithm should return two solutions. The grammar is context-free and is recognizable by a pushdown automaton (PDA). Let us recall that a PDA manipulates both a *tape*

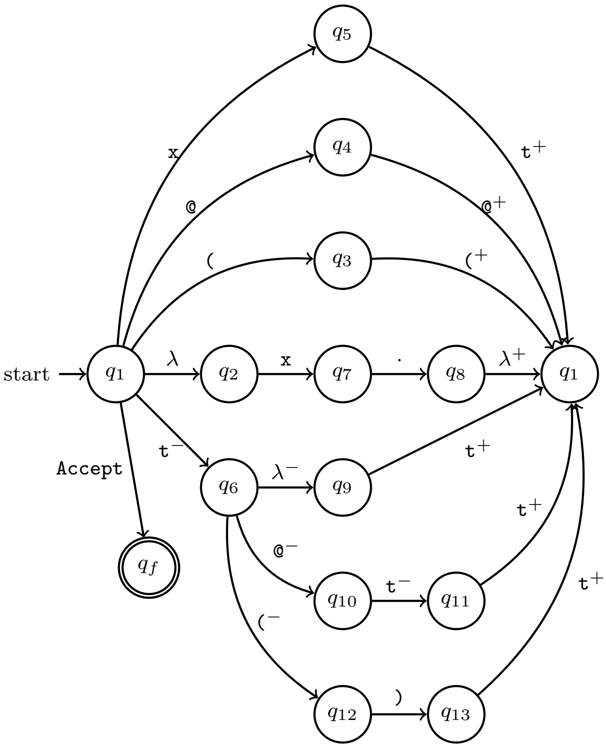
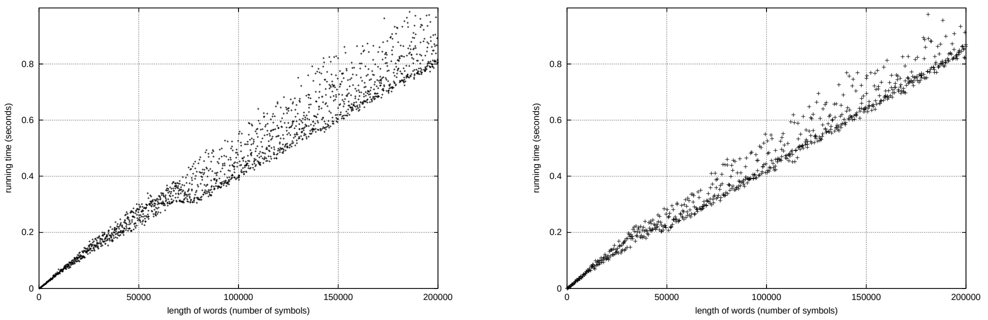


Fig. 1. Pushdown automaton (PDA) recognizing ambiguous λ-terms

and a *stack*. The word to be recognized is written on the tape and the stack is used as a weakened memory (only *push* and *pop* operations are allowed).

The PDA given in figure 1 recognizes λ-terms of the above grammar. For drawing convenience, the initial state q_1 is duplicated, its two occurrences shall be considered as equal. The stack symbols that may be encountered are the following: `t`, λ, `@` and `(`. Labels written with an exponent + (respectively −) are interpreted as a *push* (respectively a *pop*) operation on stack. Labels written without exponent are interpreted as the tape truncation. A run of this PDA is assumed to be initialized with a tape equal to the λ-term and with an empty stack. The acceptance condition of the PDA is that the tape is empty and the stack contains only the symbol `t`. The edge from q_1 to q_f labelled with `Accept` encodes this acceptance condition. In the Coq specification, the tape and stack are specified as two *`list`* values. Let *`tape`* and *`stack`* be the two corresponding datatypes, every operations labelling this PDA are easily encoded as *`relation (tape * stack)`*, thus it satisfies the first condition of finite Eilenberg machines. Also there is a measure depending on a triple of state and lengths of tape and stack which decreases along any edge; it shows that the computations in depths are necessarily finite, this is precisely the second condition that must obey a finite Eilenberg machine.

The Coq's extraction technology provides a pushdown automaton as an OCaml module *`M`* of type *`Machine`*. We obtain a reactive engine simulating the pushdown automaton plugging *`M`* to the *`Engine`* functor. Now we have a complete recognizer for words of the λ-calculus. Given a λ-term the reactive engine enumerates all possible solutions (recognitions) of the pushdown automaton. This enumeration is computed on demand as a *`stream`* value. For example, running the reactive



(a)

(b)

Fig. 2. Running time of the reactive engine for:2(a) the first solution to randomly generated ambiguous λ-terms and 2(b) the unique solution to randomly generated unambiguous λ-terms.

engine with the λ-term `"λx.x@(λx.λx.x@x)@x@x@λx.x@x"` produces a stream of length 522, that is there are 522 different ways to recognize the term; the λ-term `"x@x@x@x@x@x@x@x@x@x@x@x"` produces a stream of length 208012.

The *depth first search* strategy of the reactive engine has good complexity properties compared to a more general *breadth first search* which is not needed for simulating finite Eilenberg machines. Two benchmarks show that our approach is relevant. The first benchmark presented in figure 2(a) shows the reactive engine running time for obtaining a first solution to randomly generated λ-terms. The second benchmark presented in figure 2(b), shows the reactive engine running time for obtaining the unique solution to randomly generated *unambiguous* λ-terms (with all parentheses explicited). Both benchmarks show that the average running time of the reactive engine is linear with respect to the length of words.

8 Conclusion

We have presented a complete specification of the finite Eilenberg machines model. We have designed and formally specified a restriction of the multiset ordering [3]. It has allowed us to implement the reactive engine simulating finite Eilenberg machines into the Coq logic of total functions. Thanks to those formalizations we have been able to prove formally the correctness (soundness and completeness) of the reactive engine with regard to finite Eilenberg machines. The reactive engine presented here has taken benefit from the PROGRAM extension of Coq [10]. Its definition is very close to the one presented in OCaml in the article introducing the finite Eilenberg machine model [8].

Since deterministic and non-deterministic finite automata (DFA and NFA) and real-time transducers are particular finite Eilenberg machines, our reactive engine can be used to solve the word problem on those machines. Moreover, we have shown that our approach solves problems of higher complexity since we have embedded into the finite Eilenberg machine model a pushdown automaton recognizing ambiguous words of the λ-calculus. The reactive engine performs with success the search of all solutions in a lazy manner. The language of words of the λ-calculus belongs to the class of context-free languages. This point makes us believe that the Eilenberg machines model might offer a base for the design of a high-level language to specify and solve more general non-deterministic computational problems.

Acknowledgments

The author would like to thank Gérard Huet, Matthieu Sozeau, Jean-Baptiste Tristan and Nicolas Pouillard for their various contributions to this work.

References

[1] The Coq proof assistant. Software and documentation available on the Web, http://coq.inria.fr/, 1995–2008.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

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

[4] S. Eilenberg. *Automata, languages, and machines, Volume A*. Academic Press, 1974.

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

[6] G. Huet and B. Razet. The reactive engine for modular transducers. In J.-P. J. Kokichi Futatsugi and J. 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.

[7] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the Web, http://caml.inria.fr/, 1996–2006.

[8] B. Razet. Finite Eilenberg machines. In *13th International Conference on Implementation and Application of Automata, (CIAA 2008)*. to appear in LNCS, Springer Verlag and also available as an INRIA research report at http://hal.inria.fr/inria-00257354/, 2008.

[9] B. Razet. Simulating Eilenberg machines with a reactive engine: Formal specification, proof and program extraction. Research Report available at https://hal.inria.fr/inria-00257352, 6487, INRIA, 02 2008.

[10] M. Sozeau. Subset coercions in Coq. pages 237–252. Springer-Verlag LNCS vol. 4502, 2006.