Resumption-based big-step and small-step interpreters for While with interactive I/O

Keiko Nakata
Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia
keiko@cs.ioc.ee

In this tutorial, we program big-step and small-step total interpreters for the While language extended with input and output primitives. While is a simple imperative language consisting of skip, assignment, sequence, conditional and loop. We first develop trace-based interpreters for While. Traces are potentially infinite nonempty sequences of states. The interpreters assign traces to While programs: for us, traces are denotations of While programs. The trace is finite if the program is terminating and infinite if the program is non-terminating. However, we cannot decide (i.e., write a program to determine), for any given program, whether its trace is finite or infinite, which amounts to deciding the halting problem. We then extend While with interactive input/output primitives. Accordingly, we extend the interpreters by generalizing traces to resumptions.

The tutorial is based on our previous work with T. Uustalu on reasoning about interactive programs in the setting of constructive type theory.

1 Introduction

Interactive programs are those programs that take inputs, do some computation, output results, and iterate this cycle possibly infinitely. Operating systems and data base systems are typical examples. They are important programs and have attracted formal study to guarantee their correctness/safety. For instance, a web application should protect confidentiality of the data it processes in interaction with possibly untrusted agents, and a certified compiler should preserve input/output behavior of the source program in the compiled code. These works call for formal semantics of interactive programs.

In our previous work, we presented a constructive account of interactive input-output resumptions[1], their important properties, such as weak bisimilarity and responsiveness (a program always eventually performs input or output unless it terminates), and several coinductive operational semantics for interactive programs. Our operational semantics are resumption-based. A resumption is roughly a tree representing possible runs of a program. The tree branches on inputs, each edge corresponding to each possible input, and has infinitely deep paths if the program may diverge. The development has been fully formalized in the Coq proof assistant.

This tutorial serves to deliver the central ingredients behind our coinductive semantics by programming resumption-based interpreters for interactive programs in Haskell. No knowledge of Coq or coinduction is assumed. We use Haskell as we can naturally build and manipulate infinite objects. However, one can also work with OCaml using the lazy and force primitives.

The tutorial starts by programming trace-based interpreters for the While language. Traces, defined coinductively, are possibly infinite nonempty sequences of states. The interpreters assign traces to programs (to be more precise, to statement-state pairs), recording the states that program runs go through.

[1] The word ‘resumption’ is sometimes reserved for denotations of parallel threads. We apply it more liberally to datastructures recording evolution in small steps. This usage dates back to Plotkin [8] and was reinforced by Cenciarelli and Moggi [3].

Olivier Danvy, Chung-chieh Shan (Eds.): IFIP Working Conference
on Domain-Specific Languages 2011 (DSL 2011).
EPTCS 66, 2011, pp. 226–235, doi:10.4204/EPTCS.66.12

The trace is finite if the program run is terminating and it is infinite if the program run is non-terminating. Unlike the standard state-based interpreter for While, our trace-based interpreters are total: they return traces for *all* programs. We will develop both big-step and small-step interpreters, which are provably equivalent constructively. (The proof of the equivalence is beyond the scope of the tutorial.)

We then extend While with interactive input/output primitives and generalize traces to resumptions. Accordingly, we extend our big-step and small-step interpreters, which now assign resumptions to interactive While programs. Once one learns trace-based interpreters for While, this step is rather straightforward.

This tutorial is based on our previous work [5, 6]. The whole development in the tutorial is portable to Coq with minor syntax adjustment. We will be explicit in whether a datatype is defined inductively (`Inductive` in Coq's vernacular) or coinductively (`CoInductive`) and whether a function is defined by recursion (`Fixpoint`) or by corecursion (`CoFixpoint`), although these distinctions are not visible when one works in Haskell.

The accompanying Haskell code can be downloaded from

`http://cs.ioc.ee/~keiko/code/dsl11.tgz`.

2 Trace-based interpreters for While

States are functions from variables to values[2]. We represent both *variables* and *values* by integers:

```
type Var = Integer
type Val = Integer
type State = Var -> Val
```

Looking up a variable in a state is then simply function application:

```
lkp :: Var -> State -> Val
lkp x s = s x
```

We also define update `upd x v s` of a state `s` at a variable `x` by a value `v`:

```
upd :: Var -> Val -> State -> State
upd x v s = \y -> if x == y then v else s y
```

The syntax of arithmetic expressions is defined inductively by

```
data AExp = N Integer | V Var
          | AExp :+ AExp | AExp :- AExp | AExp :* AExp
```

The function `aexp a s` evaluates an expression `a` in a state `s`. It is defined by recursion over the syntax of expressions:

```
aexp :: AExp -> State -> Integer
aexp (N z) _       = z
aexp (V x) s       = lkp x s
aexp (a0 :+ a1) s = aexp a0 s + aexp a1 s
aexp (a0 :- a1) s = aexp a0 s - aexp a1 s
aexp (a0 :* a1) s = aexp a0 s * aexp a1 s
```

Similarly, we define boolean expressions and an evaluator for them:

[2] The Haskell code, in particular the notations, are adapted from the lecture material for a programming language semantic course by T. Uustalu.

```
data BExp = TT | FF | AExp :== AExp | AExp :<= AExp
          | Not BExp | BExp :&& BExp | BExp :|| BExp

bexp :: BExp -> State -> Bool
bexp TT _ = True
bexp FF _ = False
bexp (a0 :== a1) s = aexp a0 s == aexp a1 s
bexp (a0 :<= a1) s = aexp a0 s <= aexp a1 s
bexp (Not b) s = not (bexp b s)
bexp (a0 :&& a1) s = bexp a0 s && bexp a1 s
bexp (a0 :|| a1) s = bexp a0 s || bexp a1 s
```

The syntax of the While language is defined inductively by

```
data Stmt = Skip | Stmt :\ Stmt | Var := AExp
          | If BExp Stmt Stmt | While BExp Stmt
```

Our interpreters for While assign *trace*s to While programs. Traces are possibly infinite nonempty sequences of states. They are defined coinductively by

```
data Trace = Nil State | Delay State Trace
```

A trace may be finite or infinite. But we cannot decide, for any given trace *t*, whether *t* is finite or not. In other words, we cannot write a (total) function that returns true when the trace is finite and returns false otherwise. (Why?)

2.1 Big-step interpreter

The standard state-based interpreter for While is partial: given a statement and an initial state, it returns the final state if running the statement from the initial state is terminating. The interpreter diverges if the statement runs forever. When one works in a setting where only total functions are definable, e.g., within the logic of Coq, the state-based interpreter cannot be defined constructively, as this would require deciding the halting problem, an instance of the Principle of the Excluded Middle. Working with traces has the benefit that we do not need to decide: any statement and initial state uniquely determine some trace and we do not have to know whether this trace is finite for infinite.

Our trace-based big-step interpreter `eval` takes a statement and an initial state and returns a trace. It is defined by recursion over the syntax of statements:

```
eval :: Stmt -> State -> Trace
eval Skip s = Nil s
eval (stmt0 :\ stmt1) s = seque (eval stmt1) (eval stmt0 s)
eval (x := a) s = Delay s (Nil (upd x v s)) where v = aexp a s
eval (If b stmt0 stmt1) s =
    if bexp b s then
      Delay s (eval stmt0 s)
    else Delay s (eval stmt1 s)
eval (While b stmt0) s =
    if bexp b s then
      Delay s (loop (eval stmt0) (bexp b) s)
    else Delay s (Nil s)
```

We consider Skip to be terminal, or it does not take time to run Skip, so the interpreter returns a singleton consisting of the initial state. The trace for assignment x := a is a doubleton: it consists of the initial state and the final state obtained by updating the initial state s at x by the value of a in s. For sequence stmt0 :\ stmt1, we use an auxiliary function seque, defined by corecursion by

```
seque :: (State -> Trace) -> Trace -> Trace
seque k (Nil s) = k s
seque k (Delay s t) = Delay s (seque k t)
```

The idea is that we first run the statement stmt0 from the initial state, then stmt1 is run from the last state of the trace produced by the run of stmt0 (if the last state exists). In particular, stmt1 will not be run at all if running stmt0 from the initial state is nonterminating: then the trace for stmt0 is infinite and we never get to its last state, from where stmt1 will be run.

For conditional, the appropriate branch is run depending on whether the boolean guard evaluates to true or false. The trace contains one additional delay to the trace corresponding to the run of the branch, accounting for the time taken to evaluate the guard.

For while, we use an auxiliary function loop defined by mutual corecursion together with loopseq:

```
loop :: (State -> Trace) -> (State -> Bool) -> State -> Trace
loop k p s =
    if p s then
        case k s of
          Nil s’ -> Delay s’ (loop k p s’)
          Delay s’ t -> Delay s’ (loopseq k p t)
    else Nil s
loopseq :: (State -> Trace) -> (State -> Bool) -> Trace -> Trace
loopseq k p (Nil s) = Delay s (loop k p s)
loopseq k p (Delay s t) = Delay s (loopseq k p t)
```

The function loop takes three arguments: k for evaluating the loop body from a state; p for testing the boolean guard on a state; and a state s, which is the initial state. loopseq takes a trace, the initial trace, instead of a state, as the third argument. The two functions work as follows. loop takes care of repeating of the loop body, once the guard of a while loop has been evaluated. It analyzes the result and, if the guard is false, then the run of the loop terminates. If it is true, then the loop body is evaluated by calling k. loop then constructs the trace of the loop body by examining the result of k. If the loop body does not augment the trace, which can only happen, if the loop body is a sequence of Skips, a new round of repeating the loop body is started by a recursive call to loop. If the loop body augments the trace, the new round is reached by reconstruction of the trace of the current repetition with loopseq. On the exhaustion of this trace, loopseq recursively calls loop.

As a Haskell program, one might not find the definitions of loop and loopseq most intuitive. Indeed they are arranged so that (co)recursive calls to loop and loopseq are “guarded” by a Delay constructor. This way, Coq guarantees these functions are productive, as required by the logic of Coq.

Some design decisions we have made are that Skip does not grow a trace, so we have

```
eval Skip s = Nil s
```

But an assignment and testing the guard of an if- or while-statement contribute a state, i.e., constitute a small step, e.g., we have

```
eval (x := 17) s = Delay s (Nil (upd x 17 s))
eval (While FF Skip) s = Delay s (Nil s)
```

and

```
eval (While TT Skip) s = Delay s (Delay s (Delay s (...)))
```

This is good for several reasons. First, we have that `Skip` is the identity of sequential composition, i.e., the semantics does not distinguish *stmt*, `Skip :\` *stmt* and *stmt* `:\ Skip` for any statement *stmt*. Second, we get a notion of small steps that fully agrees with the textbook-style small-step interpreter given in the next section. The third and most important outcome is that any while-loop always progresses, because testing of the guard is a small step. Another option would be to regard testing of the guard to be instantaneous, but take leaving the loop body, or a backward jump in terms of low-level compiled code, to constitute a small step. But then we would not agree to the textbook small-step interpreter.

2.2 Small-step interpreter

We proceed to an equivalent small-step interpreter for While. It is based on an option-returning one-step reduction function `red`, defined by recursion over the syntax of statements:

```
red :: Stmt -> State -> Maybe (Stmt, State)
red Skip s = Nothing
red (x := a) s = Just (Skip, upd x v s) where v = aexp a s
red (stmt0 :\ stmt1) s =
    case red stmt0 s of
      Just (stmt0', s') -> Just (stmt0' :\ stmt1, s')
      Nothing -> red stmt1 s
red (If b stmt0 stmt1) s =
    if bexp b s then
      Just (stmt0, s)
    else Just (stmt1, s)
red (While b stmt0) s =
    if bexp b s then
      Just (stmt0 :\ While b stmt0, s)
    else Just (Skip, s)
```

The function `red` returns `Nothing` if the given statement is terminal, otherwise it one-step reduces the given statement from the given state and returns the resulting statement-state pair. Then the small-step interpreter `norm` is defined by corecursion by repeatedly calling `red`:

```
norm :: Stmt -> State -> Trace
norm stmt s =
    case red stmt s of
      Nothing -> Nil s
      Just (stmt', s') -> Delay s (norm stmt' s')
```

One can in fact prove that the big-step and small-step interpreters are equivalent: for any statement *stmt* and state *s*, `eval` *stmt s* and `norm` *stmt s* returns equal traces. The proof is found in [5], which is however beyond the scope of this tutorial.

3 Resumption-based interpreters for While with interactive I/O

We now extend While with interactive input/output primitives. The new syntax for statements is defined inductively by

```
data Stmt = Skip | Stmt :\ Stmt | Var := AExp
          | If BExp Stmt Stmt | While BExp Stmt
          | Input Var | Output AExp
```

The statement `Input x` reads an input value and stores it at the variable `x`. The statement `Output a` evaluates the expression `a` in the current state and outputs the resulting value.

To account for interactive input/output, we generalize traces to *resumptions*. Informally, a resumption is a datastructure that captures all possible evolutions of a configuration (a statement-state pair), a computation tree branching according to the external non-determinism resulting from interactive input.[3]

Resumptions are defined coinductively by

```
data Res = Ret State | In (Val -> Res) | Out (Val, Res) | Delay Res
```

so a resumption either has terminated with some final state, `Ret s`, takes an input value `v` and evolves into a new resumption `f v`, `In f`, outputs a value `v` and evolves into `r`, `Out (v, r)`, or performs an internal action (observable at best as a delay) and becomes `r`, `Delay r`.

Here are some examples of resumptions, defined by corecursion:

```
bot :: Res
bot = Delay bot
rep :: Val -> Res
rep v = Delay (Delay (Out (v, rep v)))
rep' :: Val -> Res
rep' v = Delay (Out (v, rep' v))
echo :: State -> Res
echo s = In (\v -> Delay (if v == 0 then Out (v, echo s) else Ret s))
echo' :: Res
echo' = In (\v -> Delay (if v == 0 then Out (v, echo') else bot))
```

`bot` represents a resumption that silently diverges. `rep v` outputs a value `v` forever. `rep' v` is similar but has shorter latency. Both `echo` and `echo'` echo input interactively; the former terminates when the input is 0, whereas the latter diverges in this situation.

3.1 Big-step interpreter

Extending the big-step interpreter for While to handle input/output primitives is straightforward. The new interpreter is given in figure 1. Input and output statements evaluate to corresponding resumptions that perform input or output actions and terminate thereafter. The functions `seque`, `loop` and `loopseq` are extended in an expected way.

3.2 Small-step interpreter

To define an equivalent small-step interpreter for the interactive While, we introduce labeled configurations, defined inductively by:

```
data Lconf = Ret_ State | In_ (Stmt, Val -> State)
           | Out_ (Val, Stmt, State) | Delay_ (Stmt, State)
```

[3] There are alternatives. We could have chosen to work, e.g., with functions from streams of input values into traces, i.e., computation paths.

The one-step reduction function `red` for the interactive While returns a labeled configuration, given a statement-state pair. It is defined by recursion over the syntax of statements by

```
red :: Stmt -> State -> Lconf
red Skip s = Ret_ s
red (x := a) s = Delay_ (Skip, upd x v s) where v = aexp a s
red (stmt0 :\ stmt1) s =
    case red stmt0 s of
      Ret_ s’ -> red stmt1 s’
      In_ (stmt0’, f) -> In_ (stmt0’ :\ stmt1, f)
      Out_ (v, stmt0’, s’) -> Out_ (v, stmt0’ :\ stmt1, s’)
      Delay_ (stmt0’, s’) -> Delay_ (stmt0’ :\ stmt1, s’)
red (If b stmt0 stmt1) s =
    if bexp b s then
      Delay_ (stmt0, s)
    else Delay_ (stmt1, s)
red (While b stmt0) s =
    if bexp b s then
      Delay_ (stmt0 :\ While b stmt0, s)
    else Delay_ (Skip, s)
red (Input x) s = In_ (Skip, \v -> upd x v s)
red (Output a) s = Out_ (v, Skip, s)  where v = aexp a s
```

Then the small-step interpreter is again obtained by repeatedly calling `red`. It is defined by corecursion by

```
norm :: Stmt -> State -> Res
norm stmt s =
    case red stmt s of
      Ret_ s’ -> Ret s’
      In_ (stmt’, f) -> In (\v -> norm stmt’ (f v))
      Out_ (v, stmt’, s’) -> Out (v, norm stmt’ s’)
      Delay_ (stmt’, s’) -> Delay (norm stmt’ s’)
```

3.3 Reasoning with resumptions

Resumptions are a syntax-free representation of the behavior of programs. We can reason about the behavior of programs in terms of resumptions they produce. In this subsection, we formalize two important properties of resumptions, namely responsiveness and delay bisimilarity[4]. Informally, a resumption is responsive if it always eventually performs an input or output action unless it terminates. Two resumptions are delay-bisimilar if they agree modulo finite delays.

A predicate P on resumptions is *responsive* if, whenever $P\,r$ holds, then one of the following conditions holds:

1. $r = \mathtt{Delay}^n\ (\mathtt{Res}\ s)$ for some n and s;
2. $r = \mathtt{Delay}^n\ (\mathtt{In}\ f)$ for some n and f, and for any value v, $P\,(f\,v)$;
3. $r = \mathtt{Delay}^n\ (\mathtt{Out}\ (v, r'))$ for some n and r' and $P\,r'$.

[4] We assume extensional equality on resumptions.

where the notation $\mathtt{Delay}^n\ r$ denotes $\overbrace{\mathsf{Delay}(\ldots(\mathsf{Delay}}^{n}\ r)\ldots)$.

A resumption r is responsive if there is a responsive predicate P such that $P\,r$ holds.

For instance, $\mathtt{echo}\ s$ given earlier in this section is responsive for any s, but `echo'` is not. (Take P such that, for any r, $P\,r$ holds if r is either $\mathtt{echo}\ s$, $\mathtt{Delay}\ (\mathtt{Out}\ (v,\ \mathtt{echo}\ s))$ or $\mathtt{Delay}\ (\mathtt{Ret}\ s)$.)

A binary relation R, written in infix notation, on resumptions is a *(termination-sensitive) delay-bisimulation* [10] if, whenever $r_0\,R\,r_1$ holds, then one of the following conditions holds:

1. $r_0 = \mathtt{Delay}^n\ (\mathtt{Res}\ s)$ and $r_1 = \mathtt{Delay}^{n'}\ (\mathtt{Res}\ s)$ for some n and n';
2. $r_0 = \mathtt{Delay}^n\ (\mathtt{In}\ f_0)$ and $r_1 = \mathtt{Delay}^{n'}\ (\mathtt{In}\ f_1)$ for some n, n', f_0 and f_1, and, for any value v, $(f_0\,v)\ R\ (f_1\,v)$.
3. $r_0 = \mathtt{Delay}^n\ (\mathtt{Out}\ (v,\ r'_0))$ and $r_1 = \mathtt{Delay}^{n'}\ (\mathtt{Out}\ (v,\ r'_1))$ for some n, n', r'_0 and r'_1, and $r'_0\ R\ r'_1$.
4. $r_0 = \mathtt{Delay}\ r'_0$ and $r_1 = \mathtt{Delay}\ r'_1$ for some r'_0 and r'_1, and $r'_0\ R\ r'_1$.

Two resumptions r_0 and r_1 are delay-bisimilar if there is a delay-bisimulation R such that $r_0\,R\,r_1$.

For instance, $\mathtt{rep}\ v$ and $\mathtt{rep'}\ v$ are delay-bisimilar for any v. (Take R such that, for any r and r', $r\,R\,r'$ holds if $r = \mathtt{rep}\ v$ and $r' = \mathtt{rep'}\ v$.)

4 Some further reading

The material given in this tutorial is based on [5, 6]. The former presents four trace-based coinductive operational semantics for While, big-step and small-step relational semantics and big-step and small-step functional semantics, and prove their equivalence in the constructive setting of Coq. The latter looked at a constructive account of interactive resumptions, their important properties such as delay bisimilarity and responsiveness, and gave several big-step operational semantics for interactive While.

Coinductive functional semantics similar to ones given in the tutorial have appeared in the works of J. Rutten and V. Capretta [9, 2]. J. Rutten gave in coalgebraic term a delayed state semantics for While, i.e., a semantics that, for a given statement-state pair, returns a possibly infinitely delayed state. V. Capretta also looked at a delayed state based semantics in his account of general recursion in constructive type theory. Central to him was the realization that the delay type constructor is a monad.

A general categorical account of small-step semantics has been given by I. Hasuo et al. [4].

Similar ideas also found applications in constructive formalization of domain theory, in particular to compute least upper bounds. C. Paulin-Mohring [7] developed a Coq library for constructive pointed ω-cpos and continuous functions and gave semantics for Kahn networks based on them. N. Benton et al. [1] generalized her library to treat predomains and a general lift monad, which are used to define denotational semantics for a simply-typed call-by-value lambda calculus with recursion and an untyped call-by-value lambda calculus.

5 Exercise

1. Extend While with the statement `repeat Stmt Bexp`, and adapt the interpreter accordingly.

2. Extend While with the statement `Stmt :||| Stmt`, and adapt the interpreter accordingly.
(`stmt0 :||| stmt1` non-deterministically chooses to run either of `stmt0` or `stmt1`.)

3. Write interactive programs that produce delay-bisimilar resumptions.

4. Give resumptions that are delay-bisimilar, and prove that they are indeed delay-bisimilar by finding a delay-bisimulation.

Acknowledgments The author's research was supported by the European Regional Development Fund (ERDF) through the Estonian Centre of Excellence in Computer Science (EXCS).

References

[1] N. Benton, A. Kennedy & C. Varming (2009): *Some Domain Theory and Denotational Semantics in Coq*. In S. Berghofer, T. Nipkow, C. Urban & M. Wenzel, editors: *Proc. of 22nd Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2009*, *LNCS* 5674, Springer, pp. 115–130, doi:10.1007/978-3-642-03359-9_10.

[2] V. Capretta (2005): *General recursion via coinductive types*. *Logical Methods in Comput. Sci.* 1(2), doi:10.2168/LMCS-1(2:1)2005.

[3] P. Cenciarelli & E. Moggi (1993): *A syntactic approach to modularity in denotational semantics*. In: *Proc. of 5th Biennial Meeting on Category Theory and Computer Science, CTCS 1993*.

[4] I. Hasuo, B. Jacobs & A. Sokolova (2007): *Generic trace semantics via coinduction*. *Logical Methods in Comput. Sci.* 3(4), doi:10.2168/LMCS-3(4:11)2007.

[5] K. Nakata & T. Uustalu (2009): *Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles*. In S. Berghofer, T. Nipkow, C. Urban & M. Wenzel, editors: *Proc. of 22nd Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2009*, *LNCS* 5674, Springer, pp. 375–390, doi:10.1007/978-3-642-03359-9_26.

[6] K. Nakata & T. Uustalu (2010): *Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: an exercise in mixed induction-coinduction*. In L. Aceto & P. Sobocinski, editors: *Proc. of 7th Workshop on Structural Operational Semantics, SOS 2010*, *Electron. Proc. in Theor. Comput. Sci.* 32, pp. 57–75, doi:10.4204/EPTCS.32.5.

[7] C. Paulin-Mohring (2009): *A constructive denotational semantics for Kahn networks in Coq*. In Y. Bertot, G. Huet, J.-J. Lévy & G. Plotkin, editors: *From Semantics to Computer Science – Essays in Honour of Gilles Kahn*, Cambridge University Press, pp. 383–414, doi:10.1017/CBO9780511770524.018.

[8] G. D. Plotkin (1983): *Domains ("Pisa Notes")*. Unpublished notes.

[9] J. Rutten (1999): *A note on coinduction and weak bisimilarity for While programs*. *Theor. Inform. and Appl.* 33(4–5), pp. 393–400, doi:10.1051/ita:1999125.

[10] W. P. Weijland (1989): *Synchrony and Asynchrony in Process Algebra*. Ph.D. thesis, University of Amsterdam.

```
eval :: Stmt -> State -> Res
eval Skip s = Ret s
eval (stmt0 :\ stmt1) s = seque (eval stmt1) (eval stmt0 s)
eval (x := a) s = Delay (Ret (upd x v s)) where v = aexp a s
eval (If b stmt0 stmt1) s =
    if bexp b s then
      Delay (eval stmt0 s)
    else Delay (eval stmt1 s)
eval (While b stmt0) s =
    if bexp b s then
      Delay (loop (eval stmt0) (bexp b) s)
    else Delay (Ret s)
eval (Input x) s = In (\v -> Ret (upd x v s))
eval (Output a) s = Out (v, Ret s)  where v = aexp a s

seque :: (State -> Res) -> Res -> Res
seque k (Ret s) = k s
seque k (In f) = In (\v -> seque k (f v))
seque k (Out (v, r)) = Out (v, seque k r)
seque k (Delay r) = Delay (seque k r)

loop :: (State -> Res) -> (State -> Bool) -> State -> Res
loop k p s =
    if p s then
        case k s of
          Ret s’ -> Delay (loop k p s’)
          In f -> In (\v -> loopseq k p (f v))
          Out(v, r) -> Out (v, r’) where r’ = loopseq k p r
          Delay r -> Delay r’ where r’ = loopseq k p r
    else Ret s
loopseq :: (State -> Res) -> (State -> Bool) -> Res -> Res
loopseq k p (Ret s) = Delay (loop k p s)
loopseq k p (In f) = In (\v -> loopseq k p (f v))
loopseq k p (Out(v, r)) = Out (v, loopseq k p r)
loopseq k p (Delay r) = Delay (loopseq k p r)
```

Figure 1: Resumption-based big-step interpreter for While with I/O