Effect Handlers: A Logical Perspective





Paulo Emílio de Vilhena

27th of February, 2025

Overview

In this talk, I am going to give a *high-level* but yet *extensive introduction* to *Hazel*, a *separation logic* for *effect handlers*.

Part 1. Programming

Introduction to effect handlers

- ***Live Programming***
  - *Shallow vs deep handlers*
  - *Simple examples*
  - *Advanced example:* `invert`
- ***Formal Semantics***

Part 2. Logic

Introduction to Hazel

- ***Specification Language***
- ***Reasoning Rules***
- ***Case Study***
  - *Verification of* `invert`

Part 1. *Introduction to Effect Handlers*

Effect Handlers

Effect handlers generalize *exception handlers*:

whereas *raising* an exception *discards* the computation,

performing an effect *suspends* the computation, which is reified as a *continuation*.

```
exception Division_by_zero
let ( / ) x y =
 if y = 0 then raise Division_by_zero
 else Int.div x y
let _ =
  match 1 + (1 / 0) with
  | exception Division_by_zero -> 0
  | y -> y
```

```
-: int = 0
```

```
type _ Effect.t += Division_by_zero: int t
let ( / ) x y =
 if y = 0 then perform Division_by_zero
 else Int.div x y
let _ =
  match 1 + (1 / 0) with
  | effect Division_by_zero k ->
     continue k 0
  | y -> y
```

```
-: int = 1
```

Shallow VS Deep

Effect handlers come in *two* flavors:

- *shallow handlers*, which handle the first effect; and
- *deep handlers*, which handle all the effects.

```
type _ Effect.t += E : unit t
let f () = perform E


let _ =
  shallow%match f(); f() with
  | effect E k -> continue k ()
  | y -> y
```

```
Exception: Unhandled
```

```
type _ Effect.t += E : unit t
let f () = perform E


let _ =
  match f(); f() with
  | effect E k -> continue k ()
  | y -> y
```

```
-: int = ()
```

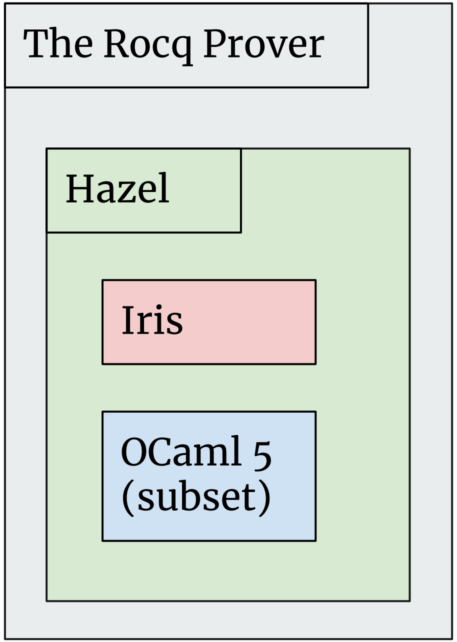
Demo!

Part 2. *Introduction to Hazel*

Specification Language

Overview of the Rocq mechanization

Hazel is an extension of *Iris*.



Iris is a modern Separation Logic:

- standard logical connectives ($\forall$, $\exists$, $\Rightarrow$, $\wedge$, $\vee$),
- separating conjunction ($*$),
- magic wand ($-\!\!*$),
- *later* modality ($\triangleright$, for *guarded recursion*),
- *persistently* modality ($\square$, to describe *duplicable resources*),
- *update* modality ($\Rrightarrow$, to support *ghost state*, a verification technique used to verify `invert`).

Formalization of the operational semantics of a subset of *OCaml 5* containing

(1) *dynamically allocated mutable state*,
(2) *effect handlers* (both *shallow* and *deep*),
(3) *global effect names* (encoded using binary sums), and
(4) *one-shot continuations*.

Protocols

In traditional Separation Logic,
a specification includes a *precondition* P and a *postcondition* Q:

$$P \mathrel{-\!\!*} \texttt{wp e } \{\texttt{y.}Q\}$$

The *key idea* of Hazel is to generalize specifications with a *protocol* Ψ,
a description of the *effects* that a program might perform.

$$P \mathrel{-\!\!*} \texttt{ewp e } \langle \Psi \rangle \; \{\texttt{y.}Q\}$$

"If the precondition P holds, then `e` *can be safely executed.*
This program either
(1) diverges, or
(2) terminates in a state where the postcondition Q holds, or
(3) performs an effect according to the protocol Ψ."

Syntax of Protocols

Ψ ::= ⊥ | !x (v) {P}. ?y (w) {Q} | Ψ + Ψ

- ***Empty protocol*** ⊥
- ***Send/recv protocol*** !x (v) {P}. ?y (w) {Q}
- ***Protocol sum*** Ψ_1 + Ψ_2

Syntax of Protocols

```
Ψ ::= ⊥ | !x (v) {P}. ?y (w) {Q} | Ψ + Ψ
```

- ***Empty protocol*** ⊥

  describes the *absence of effects*.

Examples.

```
ewp (ref 0) ⟨⊥⟩ {r. r ↦ 0}

ewp (let r = ref 1 in !r + !r) ⟨⊥⟩ {y. y = 2}
```

Syntax of Protocols

$$\Psi ::= \bot \mid \texttt{!}x\ (\texttt{v})\ \{P\}.\ \texttt{?}y\ (\texttt{w})\ \{Q\} \mid \Psi + \Psi$$

- ***Send/recv protocol*** $\texttt{!}x\ (\texttt{v})\ \{P\}.\ \texttt{?}y\ (\texttt{w})\ \{Q\}$

  attaches a *precondition* P and a *postcondition* Q to performing an effect,

  suggesting to think of *performing an effect* as *calling a function*.

"A program is allowed to perform the effect `u` *if there exists* x *such that* `u = v` *and* P *holds. For any* y *, the computation can be resumed with return value* `w` *, provided that* Q *holds."*

Syntax of Protocols

Ψ ::= $\bot$ | !x (v) {P}. ?y (w) {Q} | Ψ + Ψ

- ***Send/recv protocol*** !x (v) {P}. ?y (w) {Q}

Examples.

```
effect Abort : unit -> 'a
```

ABORT = !_ (Abort ()) {*True*}. ?y (y) {*False*}

True $-\!\!*$ ewp (perform (Abort ())) ⟨*ABORT*⟩ {_. *False*}

Syntax of Protocols

Ψ ::= $\bot$ | !x (v) {P}. ?y (w) {Q} | Ψ + Ψ

- ***Send/recv protocol*** !x (v) {P}. ?y (w) {Q}

Examples.

```
effect Get : unit -> int

GET = !x (Get ()) {currSt x}. ?_ (x) {currSt x}

currSt 1 -∗
  ewp (let x = perform (Get ()) in x + x) ⟨GET⟩
                                          {y. y = 2 * currSt 1}
```

Syntax of Protocols

$$\Psi ::= \bot \mid !x\ (v)\ \{P\}.\ ?y\ (w)\ \{Q\} \mid \Psi + \Psi$$

- ***Protocol sum*** $\Psi_1 + \Psi_2$

  describes effects that abide by *either* Ψ_1 *or* Ψ_2.

Syntax of Protocols

$$\Psi ::= \bot \mid\ !x\ (v)\ \{P\}.\ ?y\ (w)\ \{Q\} \mid \Psi + \Psi$$

- ***Protocol sum*** $\Psi_1 + \Psi_2$

Examples.

```
GET = !x   (Get ()) {currSt x}. ?_ ( x) {currSt x}
SET = !x y (Set  y) {currSt x}. ?_ (()) {currSt y}
```

```
currSt 0  -*
  ewp (let _ = perform (Set  1) in
       let x = perform (Get ()) in x + x) ⟨GET + SET⟩
                                          {y. y = 2 * currSt 1}
```

Reasoning Rules

Reasoning About Effects

(Empty)

$$\frac{\texttt{False}}{\texttt{ewp (perform u)} \ \langle \bot \rangle \ \{Q\}}$$

(Sum)

$$\frac{\texttt{ewp (perform u)} \ \langle \Psi_1 \rangle \ \{Q\} \ \vee \ \texttt{ewp (perform u)} \ \langle \Psi_2 \rangle \ \{Q\}}{\texttt{ewp (perform u)} \ \langle \Psi_1 + \Psi_2 \rangle \ \{Q\}}$$

(Send/recv)

$$\frac{\exists x.\ \texttt{u} = \texttt{v} * P * (\forall y.\ Q \twoheadrightarrow R(\texttt{w}))}{\texttt{ewp (perform u)} \ \langle !x\ (\texttt{v})\ \{P\}.\ ?y\ (\texttt{w})\ \{Q\} \rangle \ \{R\}}$$

Reasoning About Effects

(Sum)

$$\frac{\texttt{ewp (perform u)} \langle \Psi_1 \rangle \{Q\} \vee \texttt{ewp (perform u)} \langle \Psi_2 \rangle \{Q\}}{\texttt{ewp (perform u)} \langle \Psi_1 + \Psi_2 \rangle \{Q\}}$$

(Empty)

$$\frac{\mathit{False}}{\texttt{ewp (perform u)} \langle \bot \rangle \{Q\}}$$

(Send/recv)

$$\frac{\exists x.\ \texttt{u} = \texttt{v} * P * (\forall y.\ Q \twoheadrightarrow R(\texttt{w}))}{\texttt{ewp (perform u)} \langle !x\ (\texttt{v})\ \{P\}.\ ?y\ (\texttt{w})\ \{Q\} \rangle \{R\}}$$

Reasoning About Effects

(Sum)

$$\frac{\texttt{ewp (perform u)}\ \langle \Psi_1 \rangle\ \{Q\}\ \vee\quad \texttt{ewp (perform u)}\ \langle \Psi_2 \rangle\ \{Q\}}{\texttt{ewp (perform u)}\ \langle \Psi_1 + \Psi_2 \rangle\ \{Q\}}$$

(Empty)

$$\frac{\mathit{False}}{\texttt{ewp (perform u)}\ \langle \bot \rangle\ \{Q\}}$$

(Send/recv)

$$\frac{\exists x.\ \texttt{u} = \texttt{v} * P * (\forall y.\ Q \twoheadrightarrow R(\texttt{w}))}{\texttt{ewp (perform u)}\ \langle !x\ (\texttt{v})\ \{P\}.\ ?y\ (\texttt{w})\ \{Q\} \rangle\ \{R\}}$$

Reasoning About Effects

(Empty)

$$\frac{\textit{False}}{\texttt{ewp (perform u)}\ \langle \bot \rangle\ \{Q\}}$$

(Sum)

$$\frac{\texttt{ewp (perform u)}\ \langle \Psi_1 \rangle\ \{Q\} \lor \texttt{ewp (perform u)}\ \langle \Psi_2 \rangle\ \{Q\}}{\texttt{ewp (perform u)}\ \langle \Psi_1 + \Psi_2 \rangle\ \{Q\}}$$

(Send/recv)

$$\frac{\exists x.\ \texttt{u} = \texttt{v} * P * (\forall y.\ Q \twoheadrightarrow R(\texttt{w}))}{\texttt{ewp (perform u)}\ \langle !x\ (\texttt{v})\ \{P\}.\ ?y\ (\texttt{w})\ \{Q\} \rangle\ \{R\}}$$

Reasoning About Effects

(Sum)

$$\frac{\texttt{ewp (perform u)}\ \langle \Psi_1 \rangle\ \{Q\} \lor \texttt{ewp (perform u)}\ \langle \Psi_2 \rangle\ \{Q\}}{\texttt{ewp (perform u)}\ \langle \Psi_1 + \Psi_2 \rangle\ \{Q\}}$$

(Empty)

$$\frac{\mathit{False}}{\texttt{ewp (perform u)}\ \langle \bot \rangle\ \{Q\}}$$

"... is allowed to perform ... u if there exists x such that u = v and [the precondition] P holds ..."

(Send/recv)

$$\frac{\exists x.\ \texttt{u} = \texttt{v} * P * (\forall y.\ Q \twoheadrightarrow R(\texttt{w}))}{\texttt{ewp (perform u)}\ \langle !x\ (\texttt{v})\ \{P\}.\ ?y\ (\texttt{w})\ \{Q\} \rangle\ \{R\}}$$

Reasoning About Effects

(Empty)

$$\frac{\texttt{False}}{\texttt{ewp (perform u)} \langle \bot \rangle \{Q\}}$$

(Sum)

$$\frac{\texttt{ewp (perform u)} \langle \Psi_1 \rangle \{Q\} \lor \texttt{ewp (perform u)} \langle \Psi_2 \rangle \{Q\}}{\texttt{ewp (perform u)} \langle \Psi_1 + \Psi_2 \rangle \{Q\}}$$

"... for any y, the computation can be resumed with... $\texttt{w}$, provided that [the postcondition] Q holds."

(Send/recv)

$$\frac{\exists x.\ \texttt{u} = \texttt{v} * P * (\forall y.\ Q \mathrel{-\!\!*} R(\texttt{w}))}{\texttt{ewp (perform u)} \langle !x\ (\texttt{v})\ \{P\}.\ ?y\ (\texttt{w})\ \{Q\} \rangle \{R\}}$$

Local Reasoning: State

(*Frame Rule*)

$$\frac{P \mathrel{-\!\!*} \texttt{ewp e} \langle \Psi \rangle \{Q\}}{(P * R) \mathrel{-\!\!*} \texttt{ewp e} \langle \Psi \rangle \{y.\ Q(y) * R\}}$$

This is a crucial rule from *Separation Logic*.

It allows programs to be studied *separately*
if they do not alter the same data structures.

Hazel *preserves* the *frame rule*
thanks to the restriction to *one-shot continuations*.

Local Reasoning: Context

(*Bind Rule*)

$$\frac{\texttt{ewp e}\ \langle \Psi \rangle\ \{\texttt{y. ewp N[y]}\ \langle \Psi \rangle\ \{Q\}\} \qquad \texttt{N} \text{ is a } \textit{neutral context}}{\texttt{ewp N[e]}\ \langle \Psi \rangle\ \{Q\}}$$

A *neutral context* contains no handlers.

This rule allows a program to be studied *in isolation* from the context under which it is evaluated.

Reasoning About Handlers

(*Shallow Handler*)

$$\frac{\texttt{ewp e}\ \langle\Psi_1\rangle\ \{Q_1\} \qquad \mathit{isShallowHandler}\ \langle\Psi_1\rangle\ \{Q_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi_2\rangle\ \{Q_2\}}{\texttt{ewp (shallow\%match e with effect v k -> } \boldsymbol{h}\ \texttt{v k | y -> } \boldsymbol{r}\ \texttt{y)}\ \langle\Psi_2\rangle\ \{Q_2\}}$$

This rule allows the *handlee* `e` to be studied *in isolation*
from the *handler* that monitors its execution.

Intuitively, the *protocol* Ψ_1 is an abstraction boundary between *handlee* and *handler*:
performing effects is akin to *sending requests to a server*,
whose *interface* Ψ_1 the handler must *implement*.

Reasoning About Handlers

The *shallow-handler judgment* `isShallowHandler` comprises the specifications of the *return branch* and the *effect branch*:

```
isShallowHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ₂⟩ {Q₂} ≜

    (∀y. Q₁(y) -∗ ewp (r y) ⟨Ψ₂⟩ {Q₂})                    (Return branch)

  ∧

    (∀v k.                                                (Effect branch)
       ewp (perform v) ⟨Ψ₁⟩ {w.
         ewp (continue k w) ⟨Ψ₁⟩ {Q₁}
       } -∗
       ewp (h v k) ⟨Ψ₂⟩ {Q₂})
```

Reasoning About Handlers

The *shallow-handler judgment* `isShallowHandler` comprises the specifications of the *return branch* and the *effect branch*:

```
isShallowHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ₂⟩ {Q₂} ≜

    ∀y. Q₁(y) -∗ ewp (r y) ⟨Ψ₂⟩ {Q₂}

  ∧

    (∀v k.
      ewp (perform v) ⟨Ψ₁⟩ {w.
        ewp (continue k w) ⟨Ψ₁⟩ {Q₁}
      } -∗
      ewp (h v k) ⟨Ψ₂⟩ {Q₂})
```

The *return branch* can assume that `y` satisfies the handlee's *postcondition* Q_1.

Reasoning About Handlers

The *shallow-handler judgment* `isShallowHandler` comprises the specifications of the *return branch* and the *effect branch*:

$$\texttt{isShallowHandler}\ \langle\Psi_1\rangle\ \{Q_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi_2\rangle\ \{Q_2\} \triangleq$$

$$(\forall y.\ Q_1(y) \twoheadrightarrow \texttt{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{Q_2\})$$

$$\wedge$$

$$\forall v\ k.\ \ \texttt{ewp}\ (\texttt{perform}\ v)\ \langle\Psi_1\rangle\ \{w.\ \texttt{ewp}\ (\texttt{continue}\ k\ w)\ \langle\Psi_1\rangle\ \{Q_1\}\} \twoheadrightarrow \texttt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{Q_2\}$$

The *effect branch* can assume that `v` was performed under a context `k` according to the *protocol* Ψ_1.

Reasoning About Handlers

The *shallow-handler judgment* `isShallowHandler` comprises the specifications of the *return branch* and the *effect branch*:

```
isShallowHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ₂⟩ {Q₂} ≜

    (∀y. Q₁(y) -∗ ewp (r y) ⟨Ψ₂⟩ {Q₂})

  ∧

    ∀v k.
      ewp (perform v) ⟨Ψ₁⟩ {w.
        ewp (continue k w) ⟨Ψ₁⟩ {Q₁}
      } -∗
      ewp (h v k) ⟨Ψ₂⟩ {Q₂}
```

We identify the *permission* to resume the continuation.

The continuation `k` can be resumed with a return value `w`, if `w` is allowed by Ψ_1.

One is then allowed to assume that the expression `continue k w` *performs effects* according to Ψ_1 and may *terminate* according to Q_1.

Reasoning About Handlers

(*Deep Handler*)

$$\frac{\texttt{ewp e}\ \langle\Psi_1\rangle\ \{Q_1\} \qquad \mathit{isDeepHandler}\ \langle\Psi_1\rangle\ \{Q_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi_2\rangle\ \{Q_2\}}{\texttt{ewp (match e with effect v k -> } \boldsymbol{h}\texttt{ v k | v -> } \boldsymbol{r}\texttt{ v)}\ \langle\Psi_2\rangle\ \{Q_2\}}$$

The reasoning rule for *deep handlers* is similar to the rule for *shallow handlers*,
the difference is hidden in the definition of the *deep-handler judgment* `isDeepHandler`.

Reasoning About Handlers

The *deep-handler judgment* *isDeepHandler* is recursively defined, thus reflecting the recursive behavior of deep handlers.

```
isDeepHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ₂⟩ {Q₂} ≜

    (∀y. Q₁(y) −∗ ewp (r y) ⟨Ψ₂⟩ {Q₂})

  ∧

    (∀v k.
       ewp (perform v) ⟨Ψ₁⟩ {w. ∀Ψ' Q'.
         ▷ isDeepHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ'⟩ {Q'} −∗
         ewp (continue k w) ⟨Ψ'⟩ {Q'}
       } −∗
       ewp (h v k) ⟨Ψ₂⟩ {Q₂})
```

Reasoning About Handlers

The *deep-handler judgment* `isDeepHandler` is recursively defined,
thus reflecting the recursive behavior of deep handlers.

```
isDeepHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ₂⟩ {Q₂} ≜

    (∀y. Q₁(y) −∗ ewp (r y) ⟨Ψ₂⟩ {Q₂})

  ∧

    (∀v k.
       ewp (perform v) ⟨Ψ₁⟩ {w. ∀Ψ' Q'.
         ▷ isDeepHandler ⟨Ψ₁⟩ {Q₁} (h | r) ⟨Ψ'⟩ {Q'} −∗
         ewp (continue k w) ⟨Ψ'⟩ {Q'}
       } −∗
       ewp (h v k) ⟨Ψ₂⟩ {Q₂})
```

To reason about the call to the continuation, one must *reestablish* the handler judgment, because the handler is *reinstalled*.

This new handler instance may abide by a *different protocol* Ψ' and by a *different postcondition* Q'.

Case Study: Verification of `invert`

Specification of `invert`

```
type iter = (int -> unit) -> unit

type sequence = unit -> head
and head = Nil | Cons of int * sequence

val invert : iter -> sequence
```

We wish to prove that `invert` meets the following specification:

$$\forall \mathtt{iter}\ xs.\quad \mathit{isIter}(\mathtt{iter}, \mathtt{xs}) \mathrel{-\!\!*} \mathtt{ewp}\ (\mathtt{invert}\ \mathtt{iter})\ \langle \bot \rangle\ \{k.\ \mathit{isSeq}(k, \mathtt{xs})\}$$

Definition of `isIter`

```
type iter = (int -> unit) -> unit
```

$$
\begin{array}{l}
\mathit{isIter}(\mathtt{iter},\ \mathtt{xs}) \triangleq \\
\quad \forall \mathtt{f}\ I. \\
\quad\ \square\ (\forall us\ u\ vs.\ us \mathbin{+\!\!+} u :: vs = \mathtt{xs} \\
\qquad\quad I(us) \mathrel{-\!\!*} \mathtt{wp}\ (\mathtt{f}\ u)\ \{\_.\ I(us \mathbin{+\!\!+} [u])\}) \mathrel{-\!\!*} \\
\quad\ I([\,]) \mathrel{-\!\!*} \mathtt{wp}\ (\mathtt{iter}\ \mathtt{f})\ \{\_.\ I(xs)\}
\end{array}
$$

The *abstract predicate* I is the *loop invariant*:

"If `f` can take one step, then `iter` can take `xs` steps."

Definition of *isIter*

```
type iter = (int -> unit) -> unit
```

```
isIter(iter, xs) ≜
    ∀f I Ψ.
     □ (∀us u vs. us ++ u :: vs = xs −∗
        I(us) −∗ ewp (f u) ⟨Ψ⟩ {_. I(us ++ [u])}) −∗
     I([]) −∗ ewp (iter f) ⟨Ψ⟩ {_. I(xs)}
```

The *abstract predicate* I is the *loop invariant*.

The *abstract protocol* Ψ means that `iter` is *effect-polymorphic*:

(1) `iter` *does not perform* effects, and

(2) `iter` *does not intercept* the effects that `f` may throw.

Definition of `isSeq`

```
type sequence = unit -> head
and head = Nil | Cons of int * sequence
```

$$isSeq'(k, us, xs) \triangleq \mathrm{ewp}\ k()\ \langle \bot \rangle\ \{y.\ isHead(y, us, xs)\}$$

$$isHead(y, us, xs) \triangleq \texttt{match}\ y\ \texttt{with}$$

$$\quad |\ \mathrm{Nil} \Rightarrow us = xs$$

$$\quad |\ \mathrm{Cons}\ (u, k) \Rightarrow \exists vs.\ us \mathbin{++} u :: vs = xs \ \ast\ \triangleright\ isSeq'(k, us \mathbin{++} [u], xs)$$

$$\quad \texttt{end}$$

$$isSeq(k, xs) \triangleq isSeq'(k, [], xs)$$

The protocol $\bot$ indicates that a sequence *does not perform effects*.

Because the definition of `isSeq'` does not include a *persistently modality*,
the sequence `k` *is not* duplicable; it can be used *at most once*.

Key Ideas

```
type _ Effect.t += Yield : int -> unit t
let yield x = perform (Yield x)

let invert iter = fun () ->
  match iter yield with
  | effect (Yield x) k -> Seq.Cons (x, continue k)
  | ()                 -> Seq.Empty
```

We covered the definitions, now we study the *key ideas* of the proof:

1. The introduction of a piece of *ghost state* to keep track of the elements already *seen*.
2. The introduction of the protocol describing the effect `Yield`.

Ghost State

```
let yield x = perform (Yield x)

let invert iter = fun () ->
  let ghost seen = ref [] in
  match iter yield with
  | effect (Yield x), k ->
      seen := !seen @ [x];
      Seq.Cons (x, continue k)
  | () ->
      Seq.Empty
```

The memory cell `seen` is part of the *ghost state*, which can be seen as a *fictional extension of the heap*.

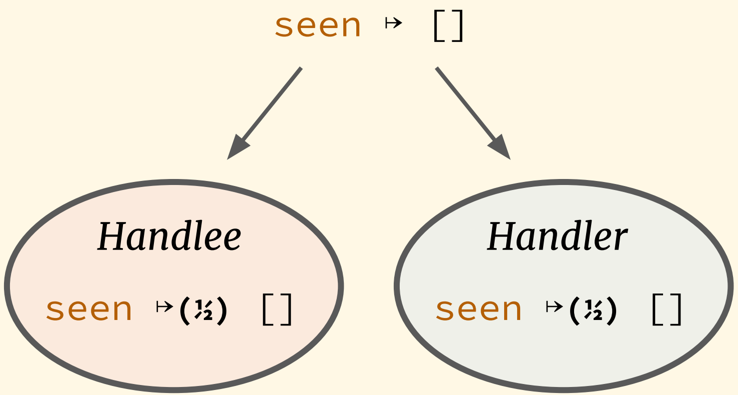
Ghost state is a standard verification technique, usually presented as *history variables*.

Ghost State

```
let yield x = perform (Yield x)

let invert iter = fun () ->
  let ghost seen = ref [] in
  match iter yield with
  | effect (Yield x), k ->
      seen := !seen @ [x];
      Seq.Cons (x, continue k)
  | () ->
      Seq.Empty
```

The *ownership* of the *ghost location* `seen` is split between *handlee* and *handler*:



To update `seen`, *full ownership* is required, which can be recovered from the *two halves*:

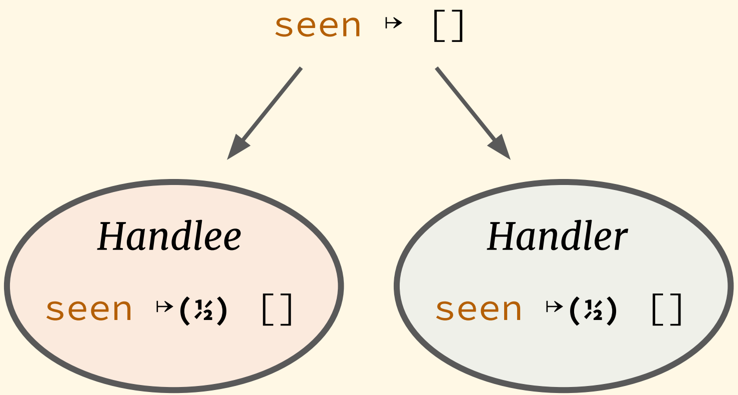
$$\mathtt{seen} \mapsto_{(1/2)} us \;-\!\!\ast\; \mathtt{seen} \mapsto_{(1/2)} vs \;-\!\!\ast\; \Rrightarrow\; \mathtt{seen} \mapsto (us \mathbin{++} [u]) \ast us = vs$$

Ghost State

```
let yield x = perform (Yield x)

let invert iter = fun () ->
  let ghost seen = ref [] in
  match iter yield with
  | effect (Yield x), k ->
     seen := !seen @ [x];
     Seq.Cons (x, continue k)
  | () ->
     Seq.Empty
```

The *ownership* of the *ghost location* seen is split between *handlee* and *handler*:



"In the eyes of the handlee, the effect Yield *u updates* seen *with u."*

```
YIELD = !us u vs (Yield u) { seen ↦(½) us          *
                             us ++ u :: vs = xs    }.
        ?_       (())      { seen ↦(½) (us ++ [u]) }
```

Verification of `invert`

```
seen ↦(½) [] -∗
ewp (iter yield) ⟨YIELD⟩ {_.
  seen ↦(½) xs}
```

```
seen ↦(½) [] -∗
isDeepHandler
  ⟨YIELD⟩ {_. seen ↦(½) xs}
    (h | r)
  ⟨⊥⟩ {y. isHead(y,[],xs)}
```

―― *(Deep Handler)*

```
seen ↦ [] -∗
ewp (match iter yield with
      | effect (Yield x) k -> h x k
      |() -> r ()) ⟨⊥⟩ {y. isHead(y,[],xs)}
```

After the allocation of `seen`, there comes the *main reasoning step*:
the application of *Rule Deep Handler*.

Verification of `invert`

First proof obligation

$$\texttt{seen} \mapsto(½)\ [\,] \;-\!\!*\; \texttt{ewp (iter yield)}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \texttt{seen} \mapsto(½)\ \mathit{xs}\}$$

The first proof obligation follows from the hypothesis *isIter*(iter, *xs*).

Indeed, it suffices

(1) to instantiate the loop invariant $I(us)$ with $\texttt{seen} \mapsto(½)\ us$,

(2) to instantiate the abstract protocol Ψ with *YIELD* , and

(2) to prove that the function `yield` *"advances the invariant by one step"*.

$$\texttt{seen} \mapsto(½)\ us \;-\!\!*\; \texttt{ewp (yield}\ u\texttt{)}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \texttt{seen} \mapsto(½)\ (us \mathbin{++} [u])\}$$

Verification of `invert`

Second proof obligation

$$\texttt{seen} \mapsto^{(\frac{1}{2})} [\,] \;-\!\!*\; \begin{array}{l} \mathit{isDeepHandler}\ \langle \mathit{YIELD}\rangle\ \{\_.\ \texttt{seen} \mapsto^{(\frac{1}{2})} xs\} \\ \quad (\mathbf{h} \mid \mathbf{r}) \\ \quad \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,[\,],xs)\} \end{array}$$

First, we generalize the assertion to reason about an arbitrary state of `seen`:

$$H \triangleq \forall us.\ \texttt{seen} \mapsto^{(\frac{1}{2})} us \;-\!\!*\; \begin{array}{l} \mathit{isDeepHandler}\ \langle \mathit{YIELD}\rangle\ \{\_.\ \texttt{seen} \mapsto^{(\frac{1}{2})} xs\} \\ \quad (\mathbf{h} \mid \mathbf{r}) \\ \quad \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,us,xs)\} \end{array}$$

The proof then follows by Löb induction (because a deep handler is recursively defined):

$$\triangleright\ H \;-\!\!*\; H$$

Demo!