Hazel: a Separation Logic for effect handlers

Paulo Emílio de Vilhena and François Pottier
What is the problem?

- We want to **formally verify** programs exploiting **effect handlers**, that is, we want to write **specifications** and **verify** that they are met.

- More specifically, we want to devise a **program logic** for effect handlers.
Why?

- **Usefulness.** To think in terms of specifications and reasoning rules is a valuable tool; formal specification provides a precise program documentation.

- **Gap.** The literature on mechanized verification methods for programs that combine effect handlers and mutable state is surprisingly scarce.
Specifying a concrete example

```plaintext

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

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

 effect Yield : int -> unit
 let yield x = perform (Yield x)

 let invert (iter : iter) : sequence =
   fun () ->
     match iter yield with
     | effect (Yield x) k ->
       Cons (x, continue k)
     | () ->
       Empty
```
A lazy sequence is a thunk that when forced will either produce a marker of its end or a pair of head and tail.

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

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

effect Yield : int -> unit
let yield x = perform (Yield x)

let invert (iter : iter) : sequence =
  fun () ->
    match iter yield with
    | effect (Yield x) k ->
      Cons (x, continue k)
    | () ->
      Empty
```
Specifying a concrete example

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

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

effect Yield : int -> unit
let yield x = perform (Yield x)

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

A higher-order iteration method is **eager**: it iterates an input function over an underlying collection of elements.
Specifying a concrete example

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

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

effect Yield : int -> unit
let yield x = perform (Yield x)

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

The function `invert` uses `Yield` to stop the iterator.
Specifying a concrete example

The intuition: \( \text{invert} \) transforms an \textit{eager} iteration method into a \textit{lazy} sequence.

```ocaml
val invert : iter -> sequence
```

Can we \textbf{state precisely} what \textit{invert} does?

1. What is \textit{iter}? (Precondition)
2. What is \textit{sequence}? (Postcondition)
3. What elements are covered by the result of \textit{invert}? (Correctness)
4. Does \textit{invert} perform effects? (Safety)
With a formal specification we can:

Formal specification

\[ \forall \text{iter } xs. \]
\[ \text{isIter iter xs} \implies \text{ewp (invert iter) } \langle \bot \rangle \{ k. \text{isSeq } k \text{ xs} \} \]

(Precondition) \text{iter iterates a given function through the elements of the list } xs.

The program \text{invert iter} can be executed, it won't perform effects and ...

... (Postcondition) if it terminates, then it returns a sequence \text{k} that produces the elements \text{xs}.
Remainder of the talk

- Presentation of Hazel. We give a broad **overview of the project** and we present the **key ideas** to **specify** and **verify** programs in our system.

- Application of Hazel. We are going to study **invert** in detail:
  - Definition of **isIter**.
  - Definition of **isSeq**.
  - Proof of **invert**.
Presentation of Hazel
Structure of the Hazel project

- **Iris.**
  A *Separation Logic*: standard logical connectives, separating conjunction (\(\land\)), magic wand (\(\leftarrow\)) and some modalities (\(\text{later} \Rightarrow\), persistent \(\Box\), etc.).

  *Why separation logic?*
  *SL provides local reasoning about the state.*

  *Why Iris?*
  *Iris is expressive, that is, many verification tasks can be carried out without ad hoc extensions.*

- **HH Programming language.**
  A subset of Multicore OCaml restricted to *Heaps* and *Handlers.*
Hazel's main feature: to generalize specifications

Traditional specification in separation logic

\[ P \text{ wp e } \{ Q \} \]

- \( P \) is the **precondition**. It must hold before the execution of the program.
- \( Q \) is the **postcondition**. It holds upon termination.
Hazel's main feature: to generalize specifications

Traditional specification in separation logic

\[ P \models^* \text{wp} \ e \ \{ Q \} \]

- \( P \) is the **precondition**. It must hold before the execution of the program.

Specification in Hazel

\[ P \models^* \text{ewp} \ e \ \{ \Psi \} \ \{ Q \} \]

- \( Q \) is the **postcondition**. It holds upon termination.

- \( \Psi \) is the **protocol**. It *describes the effects* that \( e \) might throw during its execution.
Syntax of protocols

\[ \psi ::= \bot \ | \ ! x \ (\text{Eff} \ v \ y \ P \ Q) \ | \ \psi + \psi \]

- Empty protocol: \( \bot \)

- Base protocol: \( ! x \ (\text{Eff} \ v) \ {P}. \ ?y (w) \ {Q} \)

- Protocol sum: \( \psi_1 + \psi_2 \)
Syntax of protocols

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

- Empty protocol: \( \bot \)

The empty protocol describes the \text{absence of effects}. 
Syntax of protocols

\[ \psi ::= \bot \mid !x (\text{Eff } v) \{ P \}. ?y (w) \{ Q \} \mid \psi + \psi \]

- **Empty protocol**: \( \bot \)

  The empty protocol describes the **absence of effects**.

  \[ \text{ewp } (\text{ref } 0) \langle \bot \rangle \{ r. r \mapsto 0 \} \]

  \[ \text{ewp } (\text{let } r = \text{ref } 1 \text{ in } !r + !r) \langle \bot \rangle \{ y. y = 2 \} \]

  \[ \forall \text{ iter } xs. \]
  \[ \text{isIter } \text{iter } xs \ast \]
  \[ \text{ewp } (\text{invert } \text{iter}) \langle \bot \rangle \{ k. \text{isSeq } k \text{ xs} \} \]
**Syntax of protocols**

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

- **Base protocol:** \( !x (\text{Eff } v) \{ P \}. ?y (w) \{ Q \} \)

  It captures the intuition that performing an effect can be thought of as calling a function.

  It assigns a **precondition** \( P \) and a **postcondition** \( Q \) to an effect.

  The value \( v \) is the effect argument and \( w \) is the value expected in return.

  The variables \( x \) and \( y \) are binders.

**Complete intuitive reading:**

"For every \( x \), if the program performs an effect with argument \( v \) in a state satisfying \( P \), it can expect that there exists \( y \) such that the return value is \( w \) and the state satisfies \( Q \)."
Syntax of protocols

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

- **Base protocol:** \( !x (\text{Eff } v) \{ P \}. \ ?y (w) \{ Q \} \)

  ```
  \text{effect Abort : unit} \rightarrow \text{'}a
  
  \text{ABORT} = !\_ (\text{Abort }()) \{ \text{True} \}. \ ?y (y) \{ \text{False} \}
  
  \text{True }\_\_\_\_ \text{ewp (perform (Abort ())) } \langle \text{ABORT} \rangle \{ \_ \_ \_ \_ \_ \text{False} \}
  ```
Syntax of protocols

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

- **Base protocol:** \( !x \ (Eff \ v) \ {P}. \ ?y (w) \ {Q} \)

  effect Get : unit -> int

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

  \[ currSt \ 1 \_\* \]
  \[ ewp \ (let \ x = perform \ (Get ()) \ in \ x + x) \langle \text{GET} \rangle \{y. \ y = 2 \ast \ currSt \ 1} \]
Syntax of protocols

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

- **Protocol sum:** \( \Psi_1 + \Psi_2 \)

  It describes effects that abide by **either** \( \Psi_1 \) **or** \( \Psi_2 \).

\[
\begin{align*}
\text{GET} &= !x \ (\text{Get }()) \{ \text{currSt } x \}. \ ?_\_ \ (x) \{ \text{currSt } x \} \\
\text{SET} &= !x \ y \ (\text{Set } y) \{ \text{currSt } x \}. \ ?_\_ \ (()) \{ \text{currSt } y \}
\end{align*}
\]

\[
\text{currSt } \ 0 \ \_\_\*
\]

\[
\begin{aligned}
\text{ewp} \ (\text{let } \_ = \text{perform } (\text{Set } 1) \text{ in } \\
\text{let } x = \text{perform } (\text{Get }()) \text{ in } x + x) \langle \text{GET + SET} \rangle \\
\{ y. \ y = 2 * \text{currSt } 1 \}
\end{aligned}
\]
Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

\[\text{False} \]

\[
ewp \left( \text{perform} \ (\text{Eff} \ v) \right) \langle \bot \rangle \{Q\}
\]

(Protocol-Sum-Rule)

\[
ewp \left( \text{perform} \ (\text{Eff} \ v) \right) \langle \psi_1 \ + \ \psi_2 \rangle \{Q\}
\]

(Base-Protocol-Rule)

\[
\exists x. \ v' = v * P * (\forall y. \ Q \ + \ R(w))
\]

\[
ewp \left( \text{perform} \ (\text{Eff} \ v') \right) \langle !x \ (\text{Eff} \ v) \{P\}. \ ?y \ (w) \{Q\} \rangle \{R\}
\]
Explaining protocols with reasoning rules

\[\text{(Empty-Protocol-Rule)}\]

\[\text{False} \]

\[\text{ewp (perform (Eff v))} \langle \bot \rangle \{Q\}\]

\[\text{(Protocol-Sum-Rule)}\]

\[\text{ewp (perform (Eff v))} \langle \psi_1 \rangle \{Q\} \lor \text{ewp (perform (Eff v))} \langle \psi_2 \rangle \{Q\}\]

\[\text{ewp (perform (Eff v))} \langle \psi_1 + \psi_2 \rangle \{Q\}\]

\[\text{(Base-Protocol-Rule)}\]

\[\exists x. \; v' = v \star P \star (\forall y. \; Q \star R(w))\]

\[\text{ewp (perform (Eff v'))} \langle !x (Eff v) \{P\}. \; ?y (w) \{Q\} \rangle \{R\}\]
Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

\[ \text{False} \]

\[ \text{ewp (perform (Eff v)) } \langle \bot \rangle \{ Q \} \]

(Base-Protocol-Rule)

\[ \exists x. \ v' = v \ast P \ast (\forall y. \ Q \ast R(w)) \]

\[ \text{ewp (perform (Eff v')) } \langle !x \ (Eff v) \{ P \}. \ ?y \ (w) \{ Q \} \rangle \{ R \} \]

(Protocol-Sum-Rule)

\[ \text{ewp (perform (Eff v)) } \langle \Psi_1 \rangle \{ Q \} \lor \text{ewp (perform (Eff v)) } \langle \Psi_2 \rangle \{ Q \} \]

\[ \text{ewp (perform (Eff v)) } \langle \Psi_1 + \Psi_2 \rangle \{ Q \} \]
Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

\[ \text{False} \]

\[ \text{ewp} \left( \text{perform} \left( \text{Eff} \ v \right) \right) \left( \bot \right) \{Q\} \]

(Protocol-Sum-Rule)

\[ \text{ewp} \left( \text{perform} \left( \text{Eff} \ v \right) \right) \left( \psi_1 \right) \{Q\} \lor \]

\[ \text{ewp} \left( \text{perform} \left( \text{Eff} \ v \right) \right) \left( \psi_2 \right) \{Q\} \]

\[ \text{ewp} \left( \text{perform} \left( \text{Eff} \ v \right) \right) \left( \psi_1 + \psi_2 \right) \{Q\} \]

(Base-Protocol-Rule)

\[ \exists x. \ v' = v \ast P \ast (\forall y. \ Q \ast R(w)) \]

\[ \text{ewp} \left( \text{perform} \left( \text{Eff} \ v' \right) \right) \left( !x \ (\text{Eff} \ v) \{P\}. \ ?y \ (w) \{Q\} \right) \{R\} \]
Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

\[
\begin{align*}
\text{False} \\
\text{ewp (perform (Eff v)) } \langle \bot \rangle \{Q\}
\end{align*}
\]

(Protocol-Sum-Rule)

\[
\begin{align*}
\text{ewp (perform (Eff v)) } \langle \Psi_1 \rangle \{Q\} \lor \\
\text{ewp (perform (Eff v)) } \langle \Psi_2 \rangle \{Q\} \\
\text{ewp (perform (Eff v)) } \langle \Psi_1 + \Psi_2 \rangle \{Q\}
\end{align*}
\]

(Base-Protocol-Rule)

\[
\exists x. \ v' = v * P * (\forall y. \ Q \ * R(w))
\]

\[
\text{ewp (perform (Eff v')) } \langle !x (Eff v) \{P\} \cdot ?y (w) \{Q\} \rangle \{R\}
\]

We must prove the precondition $P$.  

Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

\[
\text{False}
\]

\[
\text{ewp (perform (Eff v)) } \langle \bot \rangle \{Q\}
\]

(Protocol-Sum-Rule)

\[
\begin{align*}
\text{ewp (perform (Eff v)) } \langle \psi_1 \rangle \{Q\} & \lor \\
\text{ewp (perform (Eff v)) } \langle \psi_2 \rangle \{Q\}
\end{align*}
\]

\[
\text{ewp (perform (Eff v)) } \langle \psi_1 + \psi_2 \rangle \{Q\}
\]

(Base-Protocol-Rule)

\[
\exists x. \quad v' = v \ast P \ast (\forall y. \quad Q \ast R(w))
\]

\[
\text{ewp (perform (Eff v')) } \langle !x (Eff v) \{P\} \ast ?y(w) \{Q\} \rangle \{R\}
\]

We can assume the postcondition $Q$ to prove the continuation of the program.
Local reasoning about stateful programs

(Frame-Rule)

\[ P \rightsquigarrow ewp e \langle \psi \rangle \{Q\} \]

\[ (P * R) \rightsquigarrow ewp e \langle \psi \rangle \{y. \ Q(y) * R\} \]

- Remarks.
  This is a central rule in Separation Logic.
  It captures the intuition that different components of a software application can be analysed separately if they do not alter the same data structures.
  This rule holds in our system because we are restricted to one-shot continuations.
Context-local reasoning

(Bind-Rule)

\[\text{ewp } e \langle \psi \rangle \{y. \text{ewp } N[y] \langle \psi \rangle \{Q\}\} \quad N \text{ is a neutral context}\]

\[\text{ewp } N[e] \langle \psi \rangle \{Q\}\]

- **Remarks.**
  A neutral context does not contain handlers.

  This rule states that we can reduce the verification of a big program into simpler verification tasks.
Context-local reasoning

\[(Sequencing-Rule)\]

\[\text{ewp } e_1 \langle \psi \rangle \{\_\}. \text{ewp } e_2 \langle \psi \rangle \{Q\}\]

\[\text{ewp } (e_1 ; e_2) \langle \psi \rangle \{Q\}\]

- Remarks. We apply the Bind-Rule (with \(N := [] ; \ e_2\)) to reason about the program \((e_1 ; \ e_2)\). Notice that the protocol \(\psi\) is duplicated: a protocol in Hazel is always repetitive.
Context-local reasoning

(Sequencing-Rule)

\[
\text{ewp } e_1 \langle \Psi \rangle \{\_\}. \text{ewp } e_2 \langle \Psi \rangle \{Q\}
\]

\[
\text{ewp } (e_1 ; e_2) \langle \Psi \rangle \{Q\}
\]

- Remarks.
  We apply the Bind-Rule (with \( N := [\ ]; e_2 \)) to reason about the program \( (e_1 ; e_2) \).
  Notice that the protocol \( \Psi \) is duplicated: a protocol in Hazel is always repetitive.
Local reasoning about effectful programs

(Handler-Rule)

\[
ewp \ e \ \langle \Psi_1 \rangle \ \{ \Phi_1 \} \quad \text{ishandler} \ \langle \Psi_1 \rangle \ \{ \Phi_1 \} \ (h \ | \ r) \ \langle \Psi_2 \rangle \ \{ \Phi_2 \}
\]

\[
ewp (\text{match} \ e \ \text{with} \ \text{effect} \ (\text{Eff} \ v) \ k \rightarrow h \ v \ k \ | \ v \rightarrow r \ v) \ \langle \Psi_2 \rangle \ \{ \Phi_2 \}
\]

- Remarks.
  The client \( e \) can be verified in isolation.
  The intuition is that the protocol \( \Psi_1 \) serves as a boundary between client and handler.
Local reasoning about effectful programs

The predicate \textit{isHandler} is how we specify a handler.

\[
\text{isHandler} \left< \Psi_1 \right> \{ \Phi_1 \} (h \mid r) \left< \Psi_2 \right> \{ \Phi_2 \} \triangleq
\]

\[
(\forall y. \Phi_1(y) \Horn \text{ewp} (r \ y) \left< \Psi_2 \right> \{ \Phi_2 \}) \quad \text{(Return branch)}
\]

\wedge

\[
(\forall v \ k. \quad \text{ewp} (\text{perform} (\text{Eff} \ v)) \left< \Psi_1 \right> \{w. \ \forall \Psi' \ \Phi'\}.
\]

\[\triangleright \text{isHandler} \left< \Psi_1 \right> \{ \Phi_1 \} (h \mid r) \left< \Psi' \right> \{ \Phi' \} \quad \star
\]

\[
\text{ewp} (\text{continue} k \ w) \left< \Psi' \right> \{ \Phi' \} \quad \star
\]

\[
\text{ewp} (h \ v \ k) \left< \Psi_2 \right> \{ \Phi_2 \})
\]

(Return branch)

(Effect branch)
Local reasoning about effectful programs

The predicate `isHandler` is how we specify a handler.

\[
\text{isHandler} \langle \psi_1 \rangle \{ \phi_1 \} (h \mid r) \langle \psi_2 \rangle \{ \phi_2 \} \triangleq
\]
\[
(\forall y. \quad \phi_1(y) \rightsquigarrow \text{ewp} (r \ y) \langle \psi_2 \rangle \{ \phi_2 \})
\]
\[(\text{Return branch})\]
\[
\wedge
\]
\[
(\forall v \ k. \quad \text{ewp (perform (Eff v))} \langle \psi_1 \rangle \{ w. \ \forall \psi'. \ \phi' \}.
\]
\[
\Rightarrow \text{isHandler} \langle \psi_1 \rangle \{ \phi_1 \} (h \mid r) \langle \psi' \rangle \{ \phi' \} \quad \ast
\]
\[
\text{ewp (continue k w)} \langle \psi' \rangle \{ \phi' \} \quad \ast
\]
\[
\text{ewp (h v k)} \langle \psi_2 \rangle \{ \phi_2 \}
\]
\[(\text{Effect branch})\]
Local reasoning about effectful programs

The predicate $isHandler$ is how we specify a handler.

\[
isHandler \langle \psi_1 \rangle \{\phi_1\} (h \mid r) \langle \psi_2 \rangle \{\phi_2\} \triangleq
\]

\[
(\forall y. \phi_1(y) \implies \text{ewp} (r y) \langle \psi_2 \rangle \{\phi_2\}) \quad \text{(Return branch)}
\]
\[
\land
\]
\[
(\forall v k.
\text{ewp} (\text{perform} (\text{Eff} v)) \langle \psi_1 \rangle \{w. \forall \psi' \phi'. \}
\implies
\]
\[
isHandler \langle \psi_1 \rangle \{\phi_1\} (h \mid r) \langle \psi' \rangle \{\phi'\} \quad \ast
\]
\[
\text{ewp} (\text{continue} k w) \langle \psi' \rangle \{\phi'\}\}
\ast
\]
\[
\text{ewp} (h v k) \langle \psi_2 \rangle \{\phi_2\}
\]

\[
\text{(Effect branch)}
\]
Local reasoning about effectful programs

The predicate $isHandler$ is how we specify a handler.

\[
\text{isHandler } \langle \psi_1 \rangle \{ \phi_1 \} (h \mid r) \langle \psi_2 \rangle \{ \phi_2 \} \triangleq
\]
\[
(\forall y. \phi_1(y) \Rightarrow \text{ewp}(r y) \langle \psi_2 \rangle \{ \phi_2 \}) \quad \text{(Return branch)}
\]
\[
\wedge
\]
\[
(\forall v k. \text{ewp}(\text{perform}(\text{Eff } v)) \langle \psi_1 \rangle \{ w. \forall \psi' \phi' \}.
\]
\[
\Rightarrow \text{isHandler } \langle \psi_1 \rangle \{ \phi_1 \} (h \mid r) \langle \psi' \rangle \{ \phi' \} \Rightarrow
\]
\[
\text{ewp}(\text{continue } k w) \langle \psi' \rangle \{ \phi' \} \Rightarrow
\]
\[
\text{ewp}(h v k) \langle \psi_2 \rangle \{ \phi_2 \}) \quad \text{(Effect branch)}
\]
Local reasoning about effectful programs

The predicate isHandler is how we specify a handler.

\[
\text{isHandler } \langle \Psi_1 \rangle \{ \Phi_1 \} (h \mid r) \langle \Psi_2 \rangle \{ \Phi_2 \} \triangleq \\
(\forall y. \Phi_1(y) \Rightarrow \text{ewp } (r y) \langle \Psi_2 \rangle \{ \Phi_2 \}) \quad \text{(Return branch)}
\]
\[
\land \\
(\forall v k. \\
\begin{cases} 
\text{ewp } (\text{perform } (\text{Eff } v)) \langle \Psi_1 \rangle \{ w. \forall \Psi' \Phi'. \} \\
\quad \triangleq \text{isHandler } \langle \Psi_1 \rangle \{ \Phi_1 \} (h \mid r) \langle \Psi' \rangle \{ \Phi' \} \quad \ast \\
\quad \text{ewp } (\text{continue } k w) \langle \Psi' \rangle \{ \Phi' \} \ast \\
\text{ewp } (h v k) \langle \Psi_2 \rangle \{ \Phi_2 \}) \end{cases} \quad \text{(Effect branch)}
\]

The assumption that the client performs effects that abide by the protocol $\Psi$. 
Local reasoning about effectful programs

The predicate \textit{isHandler} is how we specify a handler.

\[
\text{isHandler} \langle \Psi_1 \rangle \{ \Phi_1 \} (h \mid r) \langle \Psi_2 \rangle \{ \Phi_2 \} \triangleq \]

\[
(\forall y. \ \Phi_1(y) \rightsquigarrow \text{ewp} (r \ y) \langle \Psi_2 \rangle \{ \Phi_2 \}) \]  

\wedge

\[
(\forall v \ k. \ \begin{cases}
\text{ewp} \ (\text{perform} \ (\text{Eff} \ v)) \langle \Psi_1 \rangle \{w. \ \forall \Psi' \ \Phi'. \} \\
\Rightarrow \text{isHandler} \langle \Psi_1 \rangle \{ \Phi_1 \} (h \mid r) \langle \Psi' \rangle \{ \Phi' \} \rightsquigarrow \\
\text{ewp} \ (\text{continue} \ k \ w) \langle \Psi' \rangle \{ \Phi' \} \rightsquigarrow \\
\text{ewp} \ (h \ v \ k) \langle \Psi_2 \rangle \{ \Phi_2 \} 
\end{cases}
\]

(Return branch)

(Effect branch)

The assumption that the client performs effects that abide by the protocol \( \Psi \).

We can identify the permission to call the continuation.
Local reasoning about effectful programs

The predicate \textit{isHandler} is how we specify a handler.

\[
\text{isHandler} \langle \Psi_1 \rangle \{ \Phi_1 \} (h \mid r) \langle \Psi_2 \rangle \{ \Phi_2 \} \triangleq \]

\[
(\forall y. \Phi_1(y) \rightsquigarrow \text{ewp}(r \ y) \langle \Psi_2 \rangle \{ \Phi_2 \}) \]

\wedge

\[
(\forall v \ k. \begin{cases}
\text{ewp}(\text{perform} (\text{Eff } v)) \langle \Psi_1 \rangle \{w. \forall \Psi' \Phi'.
\text{isHandler} \langle \Psi_1 \rangle \{ \Phi_1 \} (h \mid r) \langle \Psi' \rangle \{ \Phi' \} \triangleq \text{ewp}(\text{continue } k \ w) \langle \Psi' \rangle \{ \Phi' \}) \wedge
\text{ewp} (h \ v \ k) \langle \Psi_2 \rangle \{ \Phi_2 \})
\end{cases}
\]

The assumption that the client performs effects that abide by the protocol \(\Psi\).

The predicate \textit{isHandler} reappears as a proof obligation because a deep handler is reinstalled when we call the continuation.
Application of Hazel
Application of Hazel to the verification of invert

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

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

val invert : iter -> sequence
```

Now, we prove that invert satisfies its specification.

| Specification of invert | ∀ iter xs.  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>isIter iter xs ⊢ ewp (invert iter) ⟨⊥⟩ {k. isSeq k xs}</td>
</tr>
</tbody>
</table>
Application of Hazel to the verification of invert

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

```plaintext
isIter iter xs △

∀f I.

□ (∀us u. I(us) *) wp (f u) {_. I(us ++ [u])}) *)

I([]) * wp (iter f) {_. I(xs)}
```

The abstract predicate I is the loop invariant: "If f can take one step, then iter can take xs steps."
Application of Hazel to the verification of `invert`

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

```latex
\textbf{isIter iter xs} \triangleq \\
\forall f \ I \ \psi. \\
\square (\forall us \ u. \ I(us) \ast \ ewp (f \ u) \langle \psi \rangle \{_. \ I(us \ ++ \ [u])\}) \ast \\
I([]) \ast \ ewp (\textbf{iter} f) \langle \psi \rangle \{_. \ I(xs)\}
```

The abstract predicate $I$ is the loop invariant: "If $f$ can take one step, then $\textbf{iter}$ can take $xs$ steps."

The abstract protocol $\psi$ means that $\textbf{iter}$ is effect-polymorphic:

1. $\textbf{iter}$ does not introduce effects.
2. $\textbf{iter}$ does not handle effects that $f$ may throw.
Application of Hazel to the verification of invert

\begin{verbatim}
type sequence = unit -> head
and head = Empty | Cons of int * sequence

isSeq' k us vs \triangleq ewp (k (\langle \bot \rangle \{y. isHead y us vs\}))
isHead y us vs \triangleq match y with
 | Empty \Rightarrow vs = []
 | Cons (u, k) \Rightarrow \exists vs'. vs = u :: vs' * \triangleright isSeq' k (us ++ [u]) vs'
end

isSeq k xs \triangleright isSeq' k [] xs
\end{verbatim}

Remarks:
1. A sequence does not throw effects; it is specified under the protocol \( \bot \).
2. A sequence is ephemeral; The weakest precondition ewp is an affine assertion.
Application of Hazel to the verification of \textit{invert}

\begin{verbatim}
  effect Yield : int -> unit
  let yield x = perform (Yield x)

  let invert iter = fun () ->
      match iter yield with
      | effect (Yield x) k -> Cons (x, continue k)
      | ()                 -> Empty
\end{verbatim}

We covered the definitions, now we study the main ingredients of the proof:

1. To introduce an assertion describing the \textit{state of the handler}.
2. To introduce a protocol for the effect \textit{Yield}.
Application of Hazel to the verification of \textit{invert}

\begin{verbatim}
  effect Yield : int -> unit
  let yield x = perform (Yield x)

  let invert iter = fun () ->
    match iter yield with
      | effect (Yield x) k -> Cons (x, continue k)
      | ()                -> Empty
\end{verbatim}

We covered the definitions, now we study the main ingredients of the proof:

1. To introduce an assertion describing the \textit{state of the handler}.
2. To introduce a protocol for the effect \textit{Yield}.
Application of Hazel to the verification of invert

```
effect Yield : int -> unit
let yield x = perform (Yield x)

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

What is the state of the handler?

The state of the handler is the set of elements already seen.

The handler doesn't store these elements; there is no mutable state.

These elements are stored in a ghost cell.
The **ghost state**

(Introduce) \[ \text{True} \iff \exists \gamma. \text{clientSt}_\gamma [] \ast \text{handlerSt}_\gamma [] \]

(Confront) \[ \text{clientSt}_\gamma \text{us} \ast \text{handlerSt}_\gamma \text{vs} \ast \text{us} = \text{vs} \]

(Update) \[ \text{clientSt}_\gamma \text{us} \ast \text{handlerSt}_\gamma \text{us} \ast \ast \left\{ \begin{array}{l} \text{clientSt}_\gamma (\text{us} ++ [u]) \\ \text{handlerSt}_\gamma (\text{us} ++ [u]) \end{array} \right. \]

We can think of \( \gamma \) as a reference to the elements the handler has already seen.

The assertions \( \text{clientSt}_\gamma \text{us} \) and \( \text{handlerSt}_\gamma \text{us} \) mean the same thing: that the state of \( \gamma \) is \( \text{us} \).

\( \text{clientSt}_\gamma \) is passed to \text{iter} as the loop invariant, while \( \text{handlerSt}_\gamma \) is kept by the handler.

The handler can update \( \gamma \) only when both assertions are available.

**Note:** **ghost state** is a recurrent verification technique also known as **history variables**.
Application of Hazel to the verification of `invert`

```ocaml
effect Yield : int -> unit
let yield x = perform (Yield x)

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

We covered the definitions, now we study the main ingredients of the proof:

1. To introduce an assertion describing the *state of the handler*.
2. To introduce a protocol for the effect `Yield`. 
Application of Hazel to the verification of \textit{invert}

\begin{verbatim}
effect Yield : int -> unit
let yield x = perform (Yield x)

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

The effect \textit{Yield} \textit{u} adds one element to the set of elements seen by the handler:

\textit{YIELD} = !us u (Yield u) \{clientStγ us \}. \\
?_ (()) \{clientStγ (us ++ [u])\}
\end{verbatim}
Application of Hazel to the verification of \textit{invert}

\begin{align*}
\text{clientSt}_\gamma \begin{array}{c} \ast \\text{ewp (iter yield) } \langle \text{YIELD} \rangle \end{array} \{ \_ . \text{clientSt}_\gamma \text{xs} \}
\quad \text{handlerSt}_\gamma \begin{array}{c} \ast \end{array} \langle \text{YIELD} \rangle \{ \_ . \text{clientSt}_\gamma \text{xs} \} \\
\quad \begin{array}{c} (h \mid r) \end{array}
\quad \langle \bot \rangle \{ y . \text{isHead} \ y \ [ ] \ \text{xs} \}
\end{align*}

After unfolding some definitions we reach the heart of the proof:

\begin{align*}
\text{(Handler-Rule)}
\begin{array}{c}
\text{(clientSt}_\gamma \begin{array}{c} \ast \end{array} \text{handlerSt}_\gamma \begin{array}{c} \ast \end{array}) \ast \\
\text{ewp (match iter yield with} \end{array}
\begin{array}{c}
\mid \text{effect (Yield x) } k \to h \ x \ k \\
\mid () \to r () \end{array}
\langle \bot \rangle \{ y . \text{isHead} \ y \ [ ] \ \text{xs} \}
\end{align*}

After unfolding some definitions we reach the heart of the proof:

\begin{quote}
The claim that the handler produces a head for the complete list \text{xs}.
\end{quote}

At this point, we introduce \gamma to keep track of the state of the handler.

Then, we apply rule \textit{Handler-Rule}. 
Application of Hazel to the verification of invert

To sum up.

1. We have seen the definition of isIter.
2. We have seen the definition of isSeq.
3. We have introduced the predicates clientSt_γ and handlerSt_γ.
4. We have introduced the protocol YIELD.
5. We have considered the main step of the proof where we apply the Handler-Rule.

Remark.
Thanks to the paper "A Modular Way to Reason About Iteration" by Filliâtre and Pereira, we can generalize the specification of invert to iteration methods of arbitrary collections.
Conclusion
Conclusion

● We have introduced Hazel: a Separation Logic for effect handlers.
● The logic preserves local reasoning about stateful programs.
● The notion of protocols allows local reasoning about effectful programs.
● The logic is built on top of Iris and mechanized in Coq.
● We have seen the application of Hazel to the verification of invert.
Questions?
Application of Hazel to the verification of invert

First proof obligation

\[
\text{clientSt}_\gamma \; [] \quad \ast \\
\text{ewp} \; (\text{iter} \; \text{yield}) \; \langle \text{YIELD} \rangle \\
\{ \_; \; \text{clientSt}_\gamma \; xs \}
\]

To dispatch the first proof obligation, we specialize the assertion \( \text{isIter} \; \text{iter} \; xs \).

We instantiate the protocol \( \psi \) with \( \text{YIELD} \) and the invariant \( I \) with \( \text{clientSt}_\gamma \).

Recall the definition of \( \text{isIter} \):

\[
\text{isIter} \; \text{iter} \; xs \triangleq \forall f \; I \; \psi.
\]

\[
\square \; (\forall us \; u. \; I(us) \quad \ast \quad \text{ewp} \; (f \; u) \; \langle \psi \rangle \; \{ \_; \; I(us \; ++ \; [u]) \}) \quad \ast \\
I([]) \quad \ast \quad \text{ewp} \; (\text{iter} \; f) \; \langle \psi \rangle \; \{ \_; \; I(xs) \}
\]
Application of Hazel to the verification of invert

Second proof obligation

\[
\text{handlerSt}_\gamma \; [] \; \triangleright
\]

\[
\text{isHandler} \; \langle \text{YIELD} \rangle \; \{ \_ . \; \text{clientSt}_\gamma \; \text{xs} \}
\]

\[
(h \mid r)
\]

\[
\langle \bot \rangle \; \{ y . \; \text{isHead} \; \text{y} \; [] \; \text{xs} \}
\]

First, we generalize the assertion so that we reason about an arbitrary intermediate step of the handler, rather than the initial one:

\[
H \triangleq \forall \text{us} \; \text{vs}. \left\{ \begin{array}{c}
\text{handlerSt}_\gamma \; \text{us} \; \triangleright
\end{array} \right.
\]

\[
\text{isHandler} \; \langle \text{YIELD} \rangle \; \{ \_ . \; \text{clientSt}_\gamma \; (\text{us} \; ++ \; \text{vs})\}
\]

\[
(h \mid r)
\]

\[
\langle \bot \rangle \; \{ y . \; \text{isHead} \; \text{y} \; \text{us} \; \text{vs} \}
\]

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

\[
\triangleright \; H \; \triangleright \star \; H
\]