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

```
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

```
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.

Specifying a concrete example

```
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

```
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**: `invert` transforms an eager iteration method into a lazy sequence.

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

Can we **state precisely** what `invert` does?

1. What is `iter`? (Precondition)
2. What is `sequence`? (Postcondition)
3. What elements are covered by the result of `invert`? (Correctness)
4. Does `invert` perform effects? (Safety)

Specifying a concrete example

With a **formal specification** we can:

Formal specification

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

(Precondition) iter iterates a given function through the elements of the list xs.

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

... (Postcondition) if it terminates, then it returns a sequence k that produces the elements xs.

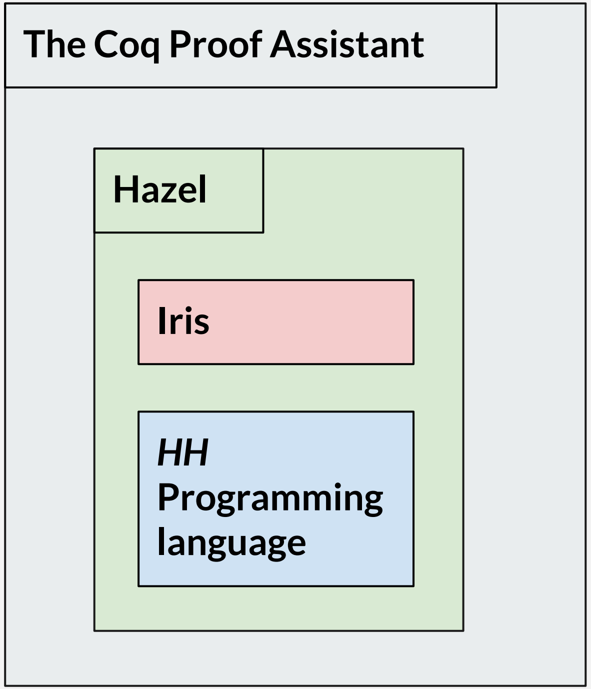
Remainder of the talk

- **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 ($*$), magic wand ($-*$) and some modalities (*later* $\triangleright$, *persistently* $\Box$, 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 *H*eaps and *H*andlers.

Hazel's main feature: to generalize specifications

Traditional specification in separation logic

$P \mathrel{-\!\!*} \mathtt{wp}\ \mathtt{e}\ \{Q\}$

- 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

$P \twoheadrightarrow \mathtt{wp}\ \mathtt{e}\ \{Q\}$

Specification in Hazel

$P \twoheadrightarrow \mathtt{ewp}\ \mathtt{e}\ \langle \Psi \rangle\ \{Q\}$

- P is the **precondition**.
  It must hold before the execution of the program.
- Q is the **postcondition**.
  It holds upon termination.
- Ψ is the **protocol**.
  It **describes the effects** that e might throw during its execution.

Syntax of protocols

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

- **Empty protocol:** `⊥`
- **Base protocol:** `!`x `(Eff v) {`P`}. ?`y `(w) {`Q`}`
- **Protocol sum:** Ψ_1 `+` Ψ_2

Syntax of protocols

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

- **Empty protocol:** $\bot$

  The empty protocol describes the absence of effects.

Syntax of protocols

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

- **Empty protocol: ⊥**

  The empty protocol describes the absence of effects.

```
ewp (ref 0) ⟨⊥⟩ {r. r ↦ 0}

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

∀ iter xs.
  isIter iter xs -∗
    ewp (invert iter) ⟨⊥⟩ {k. isSeq k xs}
```

Syntax of protocols

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

- **Base protocol:** `!`x `(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

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

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

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

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

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

Syntax of protocols

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

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

```
effect Get : unit -> int
```

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

$$currSt\ 1 \mathrel{-\!\!*}$$
$$\quad \mathtt{ewp}\ (\mathbf{let}\ \mathtt{x} = \mathtt{perform}\ (\mathtt{Get}\ ())\ \mathbf{in}\ \mathtt{x} + \mathtt{x})\ \langle GET \rangle$$
$$\qquad \{\mathtt{y}.\ \mathtt{y} = 2 * currSt\ 1\}$$

Syntax of protocols

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

- **Protocol sum:** Ψ_1 + Ψ_2

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

```
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}
```

Explaining protocols with reasoning rules

(*Empty-Protocol-Rule*)

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

(*Protocol-Sum-Rule*)

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

(*Base-Protocol-Rule*)

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

Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

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

(Protocol-Sum-Rule)

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

(Base-Protocol-Rule)

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

Explaining protocols with reasoning rules

(Protocol-Sum-Rule)

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

(Empty-Protocol-Rule)

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

(Base-Protocol-Rule)

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

Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

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

(Protocol-Sum-Rule)

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

(Base-Protocol-Rule)

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

Explaining protocols with reasoning rules

(Empty-Protocol-Rule)

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

(Protocol-Sum-Rule)

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

(Base-Protocol-Rule)

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

We must prove the precondition P.

Explaining protocols with reasoning rules

(*Empty-Protocol-Rule*)

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

(*Protocol-Sum-Rule*)

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

(*Base-Protocol-Rule*)

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

We can assume the postcondition Q to prove the continuation of the program.

Local reasoning about stateful programs

(*Frame-Rule*)

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

- **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*)

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

- **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*)

$$\frac{\texttt{ewp}\ \texttt{e}_1\ \langle\Psi\rangle\ \{\_.\ \texttt{ewp}\ \texttt{e}_2\ \langle\Psi\rangle\ \{Q\}\}}{\texttt{ewp}\ (\texttt{e}_1\ ;\ \texttt{e}_2)\ \langle\Psi\rangle\ \{Q\}}$$

- **Remarks.**
  We apply the *Bind-Rule* (with `N := []; e`$_2$) to reason about the program (`e`$_1$`;` `e`$_2$).

  Notice that the protocol Ψ is duplicated: a protocol in Hazel is always repetitive.

Context-local reasoning

(Sequencing-Rule)

$$\frac{\texttt{ewp}\ \ e_1\ \langle\Psi\rangle\ \{\_.\ \ \texttt{ewp}\ \ e_2\ \langle\Psi\rangle\ \{Q\}\}}{\texttt{ewp}\ \ (e_1\ ;\ e_2)\ \langle\Psi\rangle\ \{Q\}}$$

- **Remarks.**
  We apply the *Bind-Rule* (with $N := [\,];\ e_2$) to reason about the program $(e_1;\ e_2)$.

  Notice that the protocol Ψ is duplicated: a protocol in Hazel is always repetitive.

Local reasoning about effectful programs

(*Handler-Rule*)

$$\frac{\texttt{ewp e}\ \langle \Psi_1 \rangle\ \{\Phi_1\} \qquad \textit{isHandler}\ \langle \Psi_1 \rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle \Psi_2 \rangle\ \{\Phi_2\}}{\texttt{ewp (match e with effect (Eff v) k -> } \boldsymbol{h}\ \texttt{v k | v -> } \boldsymbol{r}\ \texttt{v)}\ \langle \Psi_2 \rangle\ \{\Phi_2\}}$$

- **Remarks.**
  The client e can be verified in isolation.

  The intuition is that the protocol Ψ_1 serves as a boundary between client and handler.

Local reasoning about effectful programs

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

$$
\begin{aligned}
&\mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\} \triangleq \\
&\qquad (\forall y.\ \Phi_1(y) \mathrel{-\!\!*} \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\}) && \textit{(Return branch)} \\
&\quad \wedge \\
&\qquad (\forall v\ k. && \textit{(Effect branch)} \\
&\qquad\quad \mathtt{ewp}\ (\mathtt{perform}\ (\mathtt{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall \Psi'\ \Phi'. \\
&\qquad\qquad \triangleright\ \mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\} \mathrel{-\!\!*} \\
&\qquad\qquad \mathtt{ewp}\ (\mathtt{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\} \mathrel{-\!\!*} \\
&\qquad\quad \mathtt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\})
\end{aligned}
$$

Local reasoning about effectful programs

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

$$\mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\}\ \triangleq$$

$$(\forall y.\ \Phi_1(y) \twoheadrightarrow \mathrm{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$ *(Return branch)*

$$\land$$

$$(\forall v\ k.$$ *(Effect branch)*

$$\mathrm{ewp}\ (\mathrm{perform}\ (\mathrm{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall \Psi'\ \Phi'.$$

$$\triangleright\ \mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\} \twoheadrightarrow$$

$$\mathrm{ewp}\ (\mathrm{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\} \twoheadrightarrow$$

$$\mathrm{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$

Local reasoning about effectful programs

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

$$\mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\} \triangleq$$

$$(\forall y.\ \Phi_1(y) \mathrel{-\!\!*} \mathrm{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\}) \qquad \textit{(Return branch)}$$

$$\land$$

$$\begin{aligned}
&(\forall v\ k. && \textit{(Effect branch)}\\
&\quad \mathrm{ewp}\ (\mathtt{perform}\ (\mathtt{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall \Psi'\ \Phi'.\\
&\quad\quad \triangleright\ \mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\} \mathrel{-\!\!*}\\
&\quad\quad \mathrm{ewp}\ (\mathtt{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\} \mathrel{-\!\!*}\\
&\quad \mathrm{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\})
\end{aligned}$$

Local reasoning about effectful programs

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

$$
\begin{array}{ll}
\mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\} \triangleq & \\
\quad (\forall y.\ \Phi_1(y) \mathrel{-\!\!*} \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\}) & \textit{(Return branch)} \\
\wedge & \\
\quad (\forall v\ k. & \textit{(Effect branch)} \\
\qquad \mathtt{ewp}\ (\mathtt{perform}\ (\mathtt{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall \Psi'\ \Phi'. & \\
\qquad\quad \triangleright\ \mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\} \mathrel{-\!\!*} & \\
\qquad\quad \mathtt{ewp}\ (\mathtt{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\} \mathrel{-\!\!*} & \\
\qquad \mathtt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\}) &
\end{array}
$$

Local reasoning about effectful programs

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

$$\mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\} \triangleq$$

$$(\forall y.\ \Phi_1(y) -\!\!* \ \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\}) \qquad \textit{(Return branch)}$$

$$\wedge$$

$$(\forall v\ k. \qquad \textit{(Effect branch)}$$

$$\left[\begin{array}{l} \mathtt{ewp}\ (\mathtt{perform}\ (\mathtt{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall\Psi'\ \Phi'. \\ \quad \triangleright\ \mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\} -\!\!* \\ \quad\ \ \mathtt{ewp}\ (\mathtt{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\} \end{array}\right. -\!\!*$$

$$\mathtt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$

The assumption that the client performs effects that abide by the protocol Ψ_1.

Local reasoning about effectful programs

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

$$\texttt{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\} \triangleq$$

$$(\forall y.\ \Phi_1(y) -\!\!*\ \texttt{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$ (Return branch)

$$\land$$

$$(\forall v\ k.$$ *(Effect branch)*

$$\texttt{ewp}\ (\texttt{perform}\ (\texttt{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall \Psi'\ \Phi'.$$

$$\triangleright\ \texttt{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\}\ -\!\!*$$

$$\texttt{ewp}\ (\texttt{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\}\ -\!\!*$$

$$\texttt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$

The assumption that the client performs effects that abide by the protocol Ψ_1.

We can identify the permission to call the continuation.

Local reasoning about effectful programs

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

$$\mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi_2\rangle\ \{\Phi_2\} \triangleq$$

$$(\forall y.\ \Phi_1(y) \mathrel{-\!\!*} \mathit{ewp}\ (\boldsymbol{r}\ y)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$

(Return branch)

$$\wedge$$

$$(\forall v\ k.$$

(Effect branch)

The assumption that the client performs effects that abide by the protocol Ψ_1.

$$\left[\begin{array}{l} \mathit{ewp}\ (\mathit{perform}\ (\mathit{Eff}\ v))\ \langle\Psi_1\rangle\ \{w.\ \forall\Psi'\ \Phi'. \\ \quad \triangleright\ \mathit{isHandler}\ \langle\Psi_1\rangle\ \{\Phi_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle\Psi'\rangle\ \{\Phi'\} \mathrel{-\!\!*} \\ \quad \mathit{ewp}\ (\mathit{continue}\ k\ w)\ \langle\Psi'\rangle\ \{\Phi'\}\} \mathrel{-\!\!*} \end{array}\right.$$

$$\mathit{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{\Phi_2\})$$

The predicate `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`

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

Application of Hazel to the verification of `invert`

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

$$\mathit{isIter}\ \mathtt{iter}\ \mathit{xs} \triangleq$$
$$\forall \mathtt{f}\ I.$$
$$\Box\,(\forall \mathit{us}\ u.\ I(\mathit{us}) \mathrel{-\!\!*} \mathtt{wp}\ (\mathtt{f}\ u)\ \{\_.\ I(\mathit{us} \mathbin{+\!\!+} [u])\}) \mathrel{-\!\!*}$$
$$I([\,]) \mathrel{-\!\!*} \mathtt{wp}\ (\mathtt{iter}\ \mathtt{f})\ \{\_.\ I(\mathit{xs})\}$$

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`

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

$$\mathit{isIter}\ \mathtt{iter}\ \mathit{xs} \triangleq$$
$$\quad \forall \mathtt{f}\ I\ \Psi.$$
$$\qquad \Box\,(\forall \mathit{us}\ u.\ I(\mathit{us}) \mathrel{-\!\!*} \mathtt{ewp}\ (\mathtt{f}\ u)\ \langle\Psi\rangle\ \{\_.\ I(\mathit{us}\ \mathtt{++}\ [u])\}) \mathrel{-\!\!*}$$
$$\qquad\quad I([]) \mathrel{-\!\!*} \mathtt{ewp}\ (\mathtt{iter}\ \mathtt{f})\ \langle\Psi\rangle\ \{\_.\ I(\mathit{xs})\}$$

The abstract predicate I is the loop invariant: *"If f can take one step, then `iter` can take `xs` steps."*

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

1. `iter` does not introduce effects.
2. `iter` does not handle effects that f may throw.

Application of Hazel to the verification of `invert`

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

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

$$\begin{array}{l} isHead\ y\ us\ vs \triangleq \mathtt{match}\ y\ \mathtt{with} \\ \quad \mid \mathtt{Empty} \Rightarrow vs = [] \\ \quad \mid \mathtt{Cons}\ (u,\ k) \Rightarrow \exists vs'.\ vs = u :: vs' * \triangleright\ isSeq'\ k\ (us \mathbin{++} [u])\ vs' \\ \quad \mathtt{end} \end{array}$$

$$isSeq\ k\ xs \triangleq isSeq'\ k\ []\ xs$$

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 `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
```

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 `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
```

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 `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*) $\quad \mathtt{True} \;\Rrightarrow\!\!\ast\; \exists\gamma.\; \mathtt{clientSt}_\gamma\; [] \ast \mathtt{handlerSt}_\gamma\; []$

(*Confront*) $\quad \mathtt{clientSt}_\gamma\; us \;-\!\!\ast\; \mathtt{handlerSt}_\gamma\; vs \;-\!\!\ast\; us = vs$

(*Update*) $\quad \mathtt{clientSt}_\gamma\; us \;-\!\!\ast\; \mathtt{handlerSt}_\gamma\; us \;\Rrightarrow\!\!\ast\; \left\{ \begin{array}{ll} \mathtt{clientSt}_\gamma & (us \mathbin{++} [u]) \\ \mathtt{handlerSt}_\gamma & (us \mathbin{++} [u]) \end{array} \right.$

We can think of γ as a reference to the elements the handler has already seen.

The assertions $\mathtt{clientSt}_\gamma\; us$ and $\mathtt{handlerSt}_\gamma\; us$ mean the same thing: that the state of γ is us.

$\mathtt{clientSt}_\gamma$ is passed to `iter` as the loop invariant, while $\mathtt{handlerSt}_\gamma$ is kept by the handler.

The handler can update γ 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`

```
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 `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
```

The effect `Yield` u adds one element to the set of elements seen by the handler:

$$\mathit{YIELD} = \;!us\; u\; (\texttt{Yield}\; u)\; \{\mathit{clientSt}_{Y}\; us\} . \;?\_\; (())\; \{\mathit{clientSt}_{Y}\; (us \mathbin{++} [u])\}$$

Application of Hazel to the verification of `invert`

$$
\dfrac{
\begin{array}{ll}
\begin{array}{l}
\mathit{clientSt}_\gamma\ [\,]\ \mathrel{-\!\!*} \\
\quad \texttt{ewp (iter yield)}\ \langle \mathit{YIELD} \rangle \\
\qquad\qquad \{\_.\ \mathit{clientSt}_\gamma\ \mathit{xs}\}
\end{array}
&
\begin{array}{l}
\mathit{handlerSt}_\gamma\ [\,]\ \mathrel{-\!\!*} \\
\quad \mathit{isHandler}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \mathit{clientSt}_\gamma\ \mathit{xs}\} \\
\qquad\qquad (\boldsymbol{h} \mid \boldsymbol{r}) \\
\qquad\quad \langle \bot \rangle\ \{y.\ \mathit{isHead}\ y\ [\,]\ \mathit{xs}\}
\end{array}
\end{array}
}{
\begin{array}{l}
(\mathit{clientSt}_\gamma\ [\,] * \mathit{handlerSt}_\gamma\ [\,])\ \mathrel{-\!\!*} \\
\quad \texttt{ewp (match iter yield with} \\
\qquad \texttt{| effect (Yield x) k ->}\ \boldsymbol{h}\ \texttt{x k} \\
\qquad \texttt{| ()}\qquad\qquad\qquad\ \texttt{->}\ \boldsymbol{r}\ \texttt{())}\ \ \langle \bot \rangle\ \{y.\ \mathit{isHead}\ y\ [\,]\ \mathit{xs}\}
\end{array}
}\ (\textit{Handler-Rule})
$$

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

The claim that the handler produces a head for the complete list xs.

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

Then, we apply rule *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 $\mathit{clientSt}_{\gamma}$ and $\mathit{handlerSt}_{\gamma}$.
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`

<u>First proof obligation</u>

$$\begin{array}{l} \mathit{clientSt}_{\gamma}\ []\ \twoheadrightarrow \\ \quad \mathtt{ewp}\ (\mathtt{iter\ yield})\ \langle \mathit{YIELD} \rangle \\ \qquad\qquad\qquad\qquad \{\_.\ \mathit{clientSt}_{\gamma}\ xs\} \end{array}$$

To dispatch the first proof obligation, we specialize the assertion $\mathit{isIter}\ \mathtt{iter}\ xs$.

We instantiate the protocol Ψ with YIELD and the invariant I with $\mathit{clientSt}_{\gamma}$.

Recall the definition of isIter:

$$\begin{array}{l} \mathit{isIter}\ \mathtt{iter}\ xs \triangleq \forall \mathtt{f}\ I\ \Psi. \\ \qquad \Box\ (\forall us\ u.\ I(us) \twoheadrightarrow \mathtt{ewp}\ (\mathtt{f}\ u)\ \langle \Psi \rangle\ \{\_.\ I(us\ {+}{+}\ [u])\})\ \twoheadrightarrow \\ \qquad\ I([])\ \twoheadrightarrow \mathtt{ewp}\ (\mathtt{iter\ f})\ \langle \Psi \rangle\ \{\_.\ I(xs)\} \end{array}$$

Application of Hazel to the verification of `invert`

Second proof obligation

$$\begin{array}{l} \mathit{handlerSt}_{\gamma}\ [\,]\ \mathrel{-\!\!*} \\ \quad \mathit{isHandler}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \mathit{clientSt}_{\gamma}\ xs\} \\ \qquad\qquad\qquad (\boldsymbol{h}\ \mid\ \boldsymbol{r}) \\ \qquad\qquad\quad \langle \bot \rangle\ \{y.\ \mathit{isHead}\ y\ [\,]\ xs\} \end{array}$$

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

$$H \triangleq \forall us\ vs.\ \left(\begin{array}{l} \mathit{handlerSt}_{\gamma}\ us\ \mathrel{-\!\!*} \\ \quad \mathit{isHandler}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \mathit{clientSt}_{\gamma}\ (us\ \mathbin{++}\ vs)\} \\ \qquad\qquad\qquad (\boldsymbol{h}\ \mid\ \boldsymbol{r}) \\ \qquad\qquad\quad \langle \bot \rangle\ \{y.\ \mathit{isHead}\ y\ us\ vs\} \end{array} \right.$$

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

$$\triangleright\ H\ \mathrel{-\!\!*}\ H$$