Proof of Programs with Effect Handlers

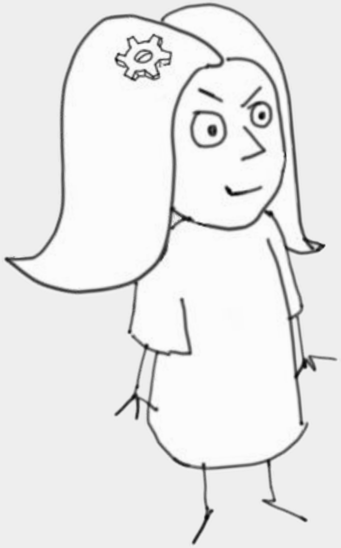
Paulo Emílio de Vilhena

16/12/2022

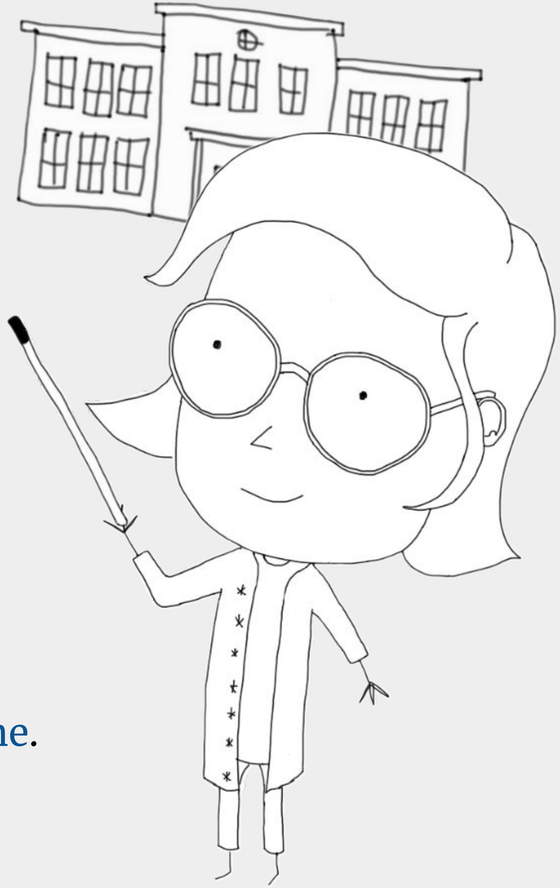
Université Paris Cité

Alice & Irene

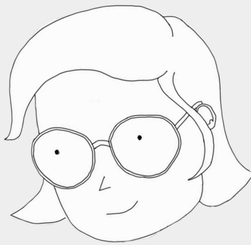
Alice is a student who is learning to program.



When she needs help,
she can count on her teacher, Irene.

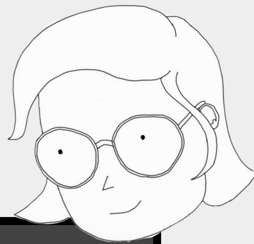


What does the function f do?



```
let rec f xs =
  match xs with
  | [] ->
     ([], [])
  | x :: xs ->
     let l, r = f xs in
     (x :: r, l)
```

What does the function `f` do?

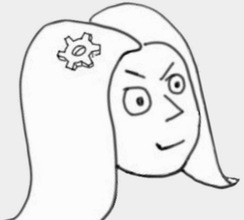


```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

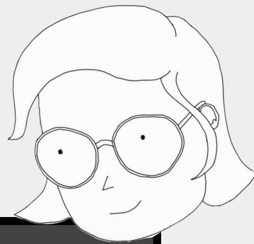
```
f [0; 1; 2];;
```

```
f [0; 1; 2; 3];;
```

```
f [0; 1; 2; 3; 4];;
```



What does the function `f` do?



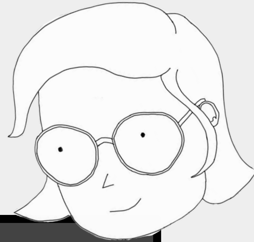
```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

```
f [0; 1; 2];;
([0; 2], [1])
```

```
f [0; 1; 2; 3];;
```

```
f [0; 1; 2; 3; 4];;
```

What does the function `f` do?

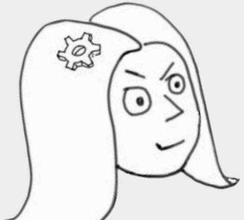


```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

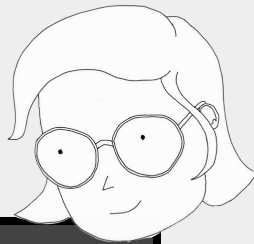
```
f [0; 1; 2];;
([0; 2], [1])
```

```
f [0; 1; 2; 3];;
([0; 2], [1; 3])
```

```
f [0; 1; 2; 3; 4];;
```



What does the function `f` do?

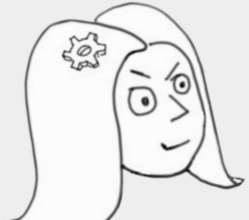


```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

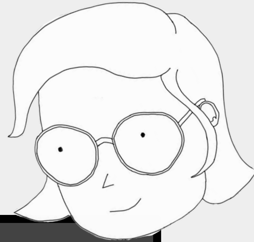
```
f [0; 1; 2];;
([0; 2], [1])
```

```
f [0; 1; 2; 3];;
([0; 2], [1; 3])
```

```
f [0; 1; 2; 3; 4];;
([0; 2; 4], [1; 3])
```



What does the function `f` do?



```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

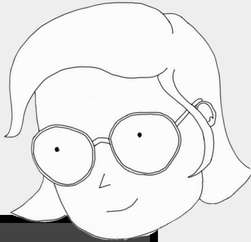
```
f [0; 1; 2];;
([0; 2], [1])
```

```
f [0; 1; 2; 3];;
([0; 2], [1; 3])
```

```
f [0; 1; 2; 3; 4];;
([0; 2; 4], [1; 3])
```

Alice: The function `f` divides the input list `xs` into a pair `(l, r)`, where `l` contains the elements at even positions and `r` contains the elements at odd positions.

What does the function f do?



```ocaml
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

```
f [0; 1; 2];;
([0; 2], [1])
```

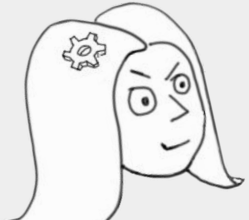
```
f [0; 1; 2; 3];;
([0; 2], [1; 3])
```

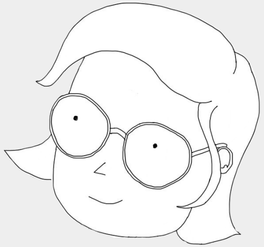
```
f [0; 1; 2; 3; 4];;
([0; 2; 4], [1; 3])
```

Alice: The function `f` divides the input list `xs` into a pair `(l, r)`, where `l` contains the elements at even positions and `r` contains the elements at odd positions.

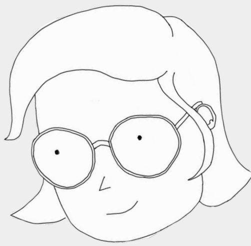
Why does `f` swap the lists `l` and `r`?

```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```





Why does `f` swap the lists `l` and `r`?



```
let rec f xs =
  match xs with
  | [] ->
    ([], [])
  | x :: xs ->
    let l, r = f xs in
    (x :: r, l)
```

Irene: Let us introduce the following notation.

evens`(xs)` ≜ *"The elements of* `xs` *at even positions."*

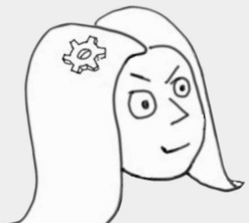
odds`(xs)` ≜ *"The elements of* `xs` *at odd positions."*

Irene: Now, look, the elements at *even* positions of `x::xs` consist of `x` plus the elements of `xs` at *odd* positions!

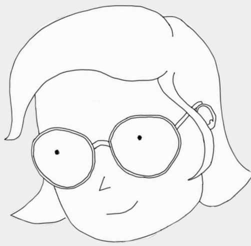
Irene: And the elements at *odd* positions of `x::xs` are the elements of `xs` at *even* positions!

evens`(x :: xs) = x ::` *odds*`(xs) = x :: r`

odds`(x :: xs) =` *evens*`(xs) = l`



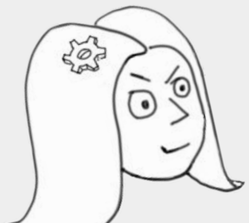
Two objections



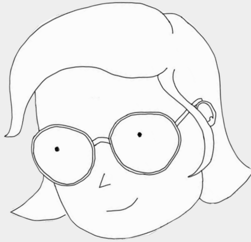
Although Irene's explanations were very helpful, Alice had two objections.

- Alice: There seems to be a kind of *circular reasoning*:
  When reasoning about the correctness of `f`,
  we assumed that the *recursive calls* to `f` behave correctly... what justifies this assumption?

- Alice: This program runs in a *computer*,
  whereas this explanation was written in the *blackboard* ...
  Why is it sound to reason about `f` independently of the machine?



Operational reasoning

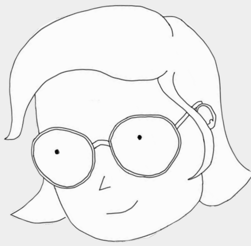


Irene: When reasoning about f on the blackboard, we relied on *operational semantics*, a formalization of the meaning of programs that is *independent* of the machine!

Irene: With *op. semantics*, one can *reason* about the relation between a program `e` and its result `v`:

$$e \rightarrow^* v$$

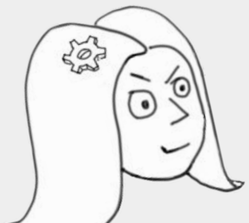
Operational reasoning



Irene: For example, we can formally express what `f` does.

```
f xs →* (evens(xs),odds(xs))
```

Irene: And we can prove this statement by *induction* on `xs`,
thus justifying that the recursive calls to `f` behave correctly.



Limitations of operational reasoning

Alice decided to apply operational reasoning to study this implementation of *fast exponentiation*.

```
let pow x n =
  let r, b = ref 1, ref x in
  let rec step k =
    if k <> 0 then begin
      if k mod 2 <> 0 then
        r := !r * !b;
      b := !b * !b;
      step (k / 2)
    end
  in
  step n; !r
```

She started by writing what `pow` does:

$$\texttt{pow x n} \quad \rightarrow^{*} \quad x^{n}$$

Limitations of operational reasoning

Alice decided to apply operational reasoning to study this implementation of *fast exponentiation*.

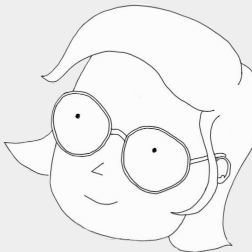
```
let pow x n =
  let r, b = ref 1, ref x in
  let rec step k =
    if k <> 0 then begin
      if k mod 2 <> 0 then
        r := !r * !b;
      b := !b * !b;
      step (k / 2)
    end
  in
  step n; !r
```

She started by writing what pow does:

$$\cancel{\texttt{pow x n} \quad \to^* \quad x^n}$$

Irene: pow has an action on the *global state*.

$$(\texttt{pow x n}, \sigma) \quad \to^* \quad (x^n, \sigma[\_:=\_][\_:=\_])$$



Limitations of operational reasoning

Alice decided to apply operational reasoning to study this implementation of *fast exponentiation*.

```
let pow x n =
  let r, b = ref 1, ref x in
  let rec step k =
    if k <> 0 then begin
      if k mod 2 <> 0 then
        r := !r * !b;
      b := !b * !b;
      step (k / 2)
    end
  in
  step n; !r
```

Alice then began to study the function `step`.
However, Alice forgot to reason by *induction*...
she would continuously unfold the definition of `step` at each recursive call.

`(step k,_) →* (step (k / 2),_) →* (step (k / 4),_) →*…`

Limitations of operational reasoning

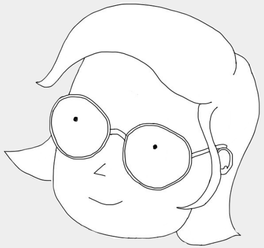
Irene: This example shows *two limitations* of operational reasoning.

```
let pow x n =
  let r, b = ref 1, ref x in
  let rec step k =
    if k <> 0 then begin
      if k mod 2 <> 0 then
        r := !r * !b;
      b := !b * !b;
      step (k / 2)
    end
  in
  step n; !r
```

Operational reasoning is *cumbersome*,
one must reason about the *global state*.

Operational reasoning is *dangerous*,
nothing stops one from indefinitely exploring the operational behavior of a given program.

Irene: Let me teach you a reasoning tool that overcomes both limitations!



Logical reasoning

Separation Logic comprises a *specification language* and a set of *reasoning rules*.

Specifications

`{P} e {y.Q}`

- *Precondition P*
  describes the state before executing `e`.

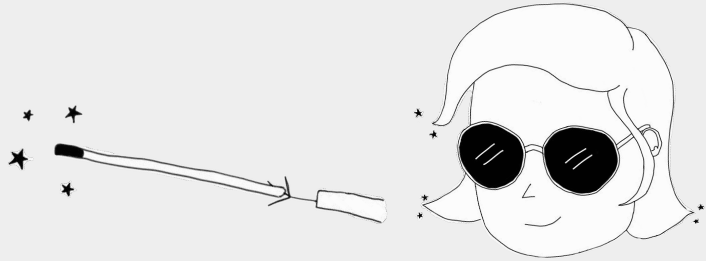
- *Postcondition Q*
  describes the state after executing `e`.

Reasoning rules

$$\frac{(\forall x.\ \{P\}\ f\ x\ \{y.Q\}) \Rightarrow \{P\}\ e\ \{y.Q\}}{\{P\}\ \textbf{let rec}\ f\ x = e\ \textbf{in}\ f\ x\ \{y.Q\}}$$

- *General rules to compose and derive specifications.*
  Taken as axioms or proven once and for all.

Convenience of logical reasoning



With *Separation Logic*,
one can reason about the functions `pow` and `step` with ease.

```
let pow x n =
  let r, b = ref 1, ref x in
  let rec step k =
    if k <> 0 then begin
      if k mod 2 <> 0 then
        r := !r * !b;
      b := !b * !b;
      step (k / 2)
    end
  in
  step n; !r
```

The specification of `pow` *hides the state*,
`pow` is *apparently pure*:

$\{n >= 0\}$ `pow x n` $\{y.\ y = x^n\}$

The specification of `step`
(1) mentions the *relevant portions of memory*
(2) includes *non-aliasing assumptions*.

$\{\ r \mapsto a \ast b \mapsto x\ \}$
`step k`
$\{\_.\ r \mapsto (a \cdot x^k) \ast b \mapsto \_\ \}$

The goal of this thesis

This is the end of Alice & Irene's story ... but a new chapter might be out soon!

Separation Logic also has limitations: it offers no way to reason about *effect handlers*.

The goal of this thesis is to extend *Separation Logic* with support for this feature.

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
```

```
effect Division_by_zero : int
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
```

Effect handlers

Effect handlers come in *two* flavors:

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

```
effect E : unit
let f () = perform E


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

```
Exception: Unhandled
```

```
effect E : unit
let f () = perform E


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

```
-: unit = ()
```

Applications *of effect handlers*

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*

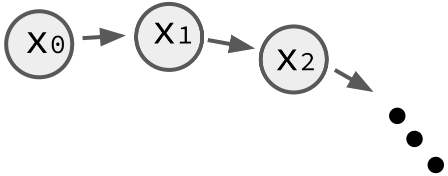
- *Asynchronous computation.*

Applications of effect handlers

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*



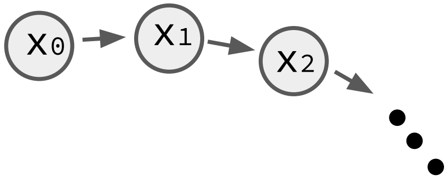
- *Asynchronous computation.*

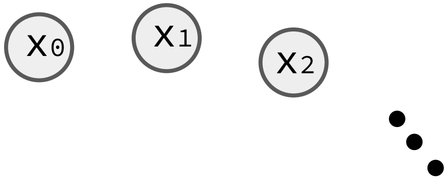
Applications of effect handlers

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*





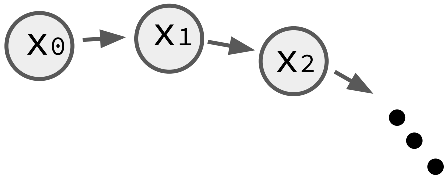
- *Asynchronous computation.*

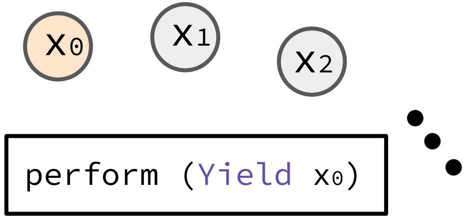
Applications of effect handlers

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*





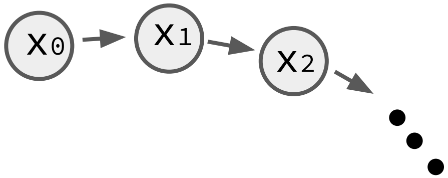
- *Asynchronous computation.*

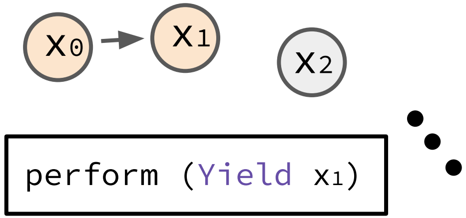
Applications *of effect handlers*

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*





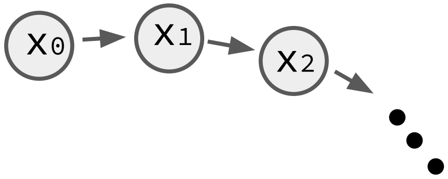
- *Asynchronous computation.*

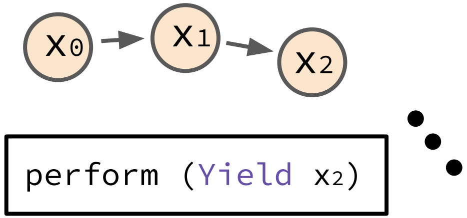
Applications of effect handlers

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*





- *Asynchronous computation.*

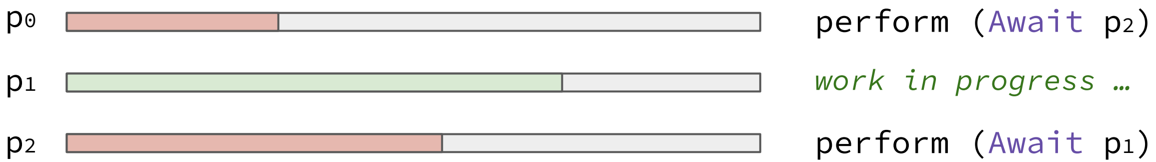
Applications of effect handlers

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

- *Control inversion.*

- *Asynchronous computation.*



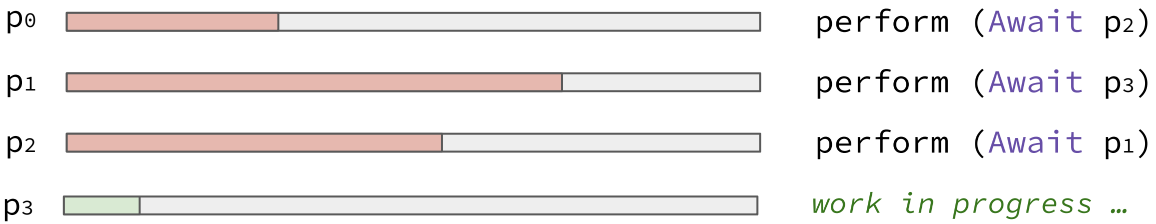
Applications of effect handlers

The ability to *suspend* a computation and *resume* it at a later time is *extremely powerful*.

There are multiple important *applications* of effect handlers:

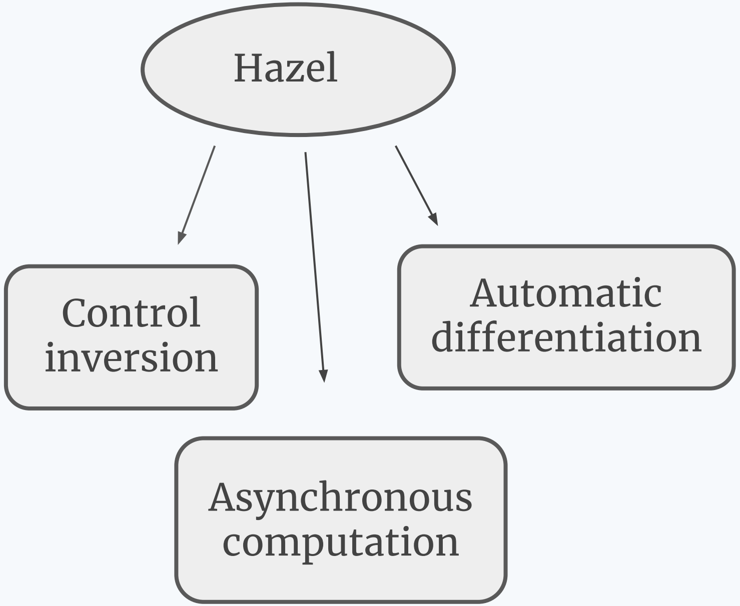
- *Control inversion.*

- *Asynchronous computation.*



Contributions of this thesis

This thesis introduces *Hazel*, a *Separation Logic* for effect handlers.



Hazel

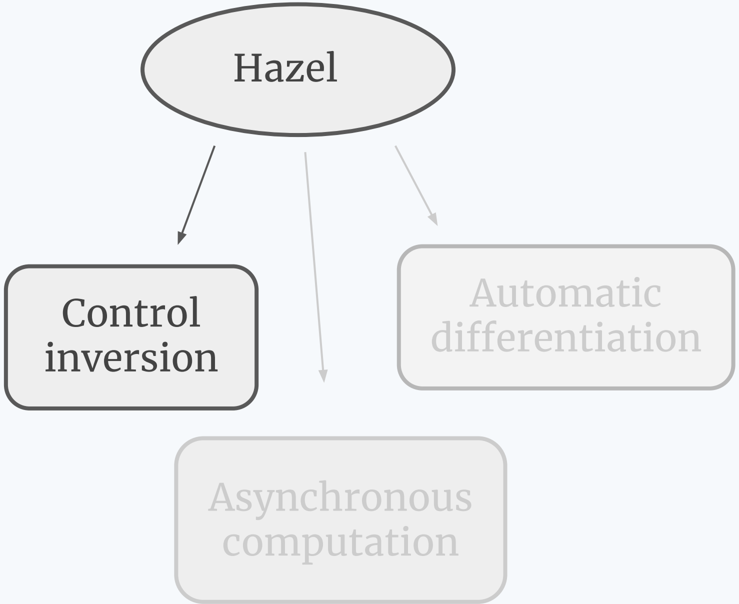
- allows *specification* and *verification* of handlers,
- preserves modular reasoning about the *state*,
- enforces new forms of modular reasoning:
  handlee (program that *performs* effects) vs
  handler (program that *handles* effects).

The *applicability* of Hazel is assessed by a number of interesting *case studies*:

- *Control inversion*
- *Asynchronous computation*
- *Automatic differentiation*

Contributions of this thesis

In the next part of the talk, I present *Hazel* and explain its application to *control inversion*.



Hazel

- allows *specification* and *verification* of handlers,
- preserves modular reasoning about the *state*,
- enforces new forms of modular reasoning:
  handlee (program that *performs* effects) vs
  handler (program that *handles* effects).

The *applicability* of Hazel is assessed by
a number of interesting *case studies*:

- *Control inversion*
- *Asynchronous computation*
- *Automatic differentiation*

Control inversion

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


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


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)
    | () ->
        Nil
```

Control inversion

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


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


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)
    | () ->
        Nil
```

A higher-order iteration method is eager: it iterates an input function over the elements of a collection.

Control inversion

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


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


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)
    | () ->
        Nil
```

A lazy sequence is a thunk that when forced produces either a marker of its end or a pair of head and tail.

Control inversion

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


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


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)
    | () ->
        Nil
```

The function `invert` *transforms an iteration method into a sequence.*

From a high-level point of view, the function `invert` *exploits an effect* `Yield` *to stop the iteration.*

Teaser - Specification of `invert` in Hazel

The behavior of `invert` is concisely specified in Hazel:

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

- *Precondition* $\mathit{isIter}(\mathtt{iter}, \mathtt{xs})$
  states that `iter` is an *iteration method* for the elements `xs`.

- *Postcondition* $\mathit{isSeq}(\mathtt{k}, \mathtt{xs})$
  states that `invert` produces a *sequence* `k` that covers the same set of elements `xs`.

- *Protocol* $\bot$
  states that `invert` does not perform unhandled effects.

Remainder of this presentation

After the introduction of Hazel, we apply this tool to reason about `invert` and prove this spec:

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

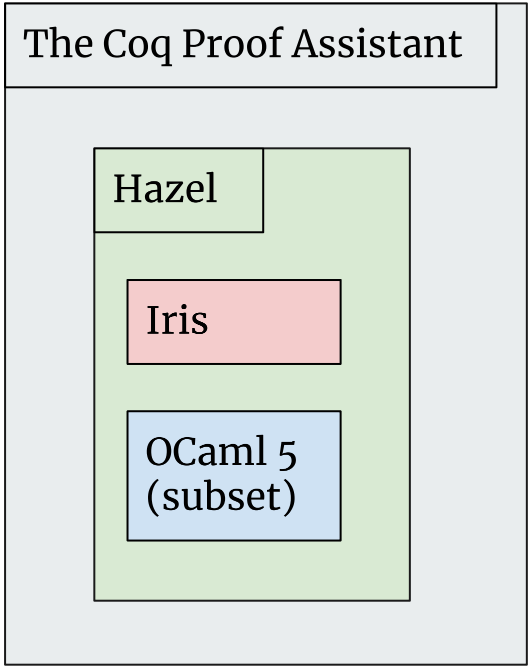
In particular, we are going to introduce

- *The notion of protocols*
- *The definition of* $\mathit{isIter}(\texttt{iter}, \texttt{xs})$
- *The definition of* $\mathit{isSeq}(\texttt{k}, \texttt{xs})$

Hazel

Overview of the Hazel project

Hazel is an extension of *Iris*.



Iris is a modern Separation Logic:

- standard logical connectives ($\forall$, $\exists$, $\Rightarrow$, $\wedge$, $\vee$),
- separating conjunction ($\ast$),
- magic wand ($-\!\!\ast$),
- *later* modality ($\triangleright$, for *guarded recursion*),
- *persistently* modality ($\square$, to describe *duplicable resources*),
- *update* modality ($\mid\!\!\Rrightarrow$, 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 `::=` $\bot$ `|` `!`x `(v)` `{`P`}.` `?`y `(w)` `{`Q`}` `|` Ψ `+` Ψ

- ***Empty protocol*** $\bot$
- ***Send/recv protocol*** `!`x `(v)` `{`P`}.` `?`y `(w)` `{`Q`}`
- ***Protocol sum*** $\Psi_1 + \Psi_2$

Syntax of protocols

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

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

  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

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

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

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

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

Examples.

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

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

$True \twoheadrightarrow \texttt{ewp}\ (\texttt{perform}\ (\texttt{Abort}\ ()))\ \langle ABORT \rangle\ \{\_.\ False\}$

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

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

- ***Protocol sum*** Ψ_1 + Ψ_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)

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

(Sum)

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

(Send/recv)

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

Reasoning about effects

(Empty)

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

(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

(*Empty*)

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

(*Sum*)

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

(*Send/recv*)

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

Reasoning about effects

(*Empty*)

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

(*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

(Empty)

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

(Sum)

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

"... 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)

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

(Sum)

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

"... 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 about state

(Frame Rule)

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

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*.

Context-local reasoning

(Bind Rule)

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

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*)

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

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*:

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

$$(\forall y.\ Q_1(y) \twoheadrightarrow \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle \Psi_2 \rangle\ \{Q_2\}) \qquad \textit{(Return branch)}$$

$$\land$$

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

$$\mathtt{ewp}\ (\mathtt{perform}\ v)\ \langle \Psi_1 \rangle\ \{w.$$

$$\mathtt{ewp}\ (\mathtt{continue}\ k\ w)\ \langle \Psi_1 \rangle\ \{Q_1\}$$

$$\} \twoheadrightarrow$$

$$\mathtt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle \Psi_2 \rangle\ \{Q_2\})$$

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*:

```
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 *effect branch* can assume that `v` was performed under a context (represented by) `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*)

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

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.

$$
\begin{aligned}
&\mathit{isDeepHandler}\ \langle \Psi_1 \rangle\ \{Q_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle \Psi_2 \rangle\ \{Q_2\} \triangleq \\
&\qquad (\forall y.\ Q_1(y) \twoheadrightarrow \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle \Psi_2 \rangle\ \{Q_2\}) \\
&\quad \wedge \\
&\qquad (\forall v\ k. \\
&\qquad\quad \mathtt{ewp}\ (\mathtt{perform}\ v)\ \langle \Psi_1 \rangle\ \{w.\ \forall \Psi'\ Q'. \\
&\qquad\qquad \triangleright\ \mathit{isDeepHandler}\ \langle \Psi_1 \rangle\ \{Q_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle \Psi' \rangle\ \{Q'\} \twoheadrightarrow \\
&\qquad\qquad \mathtt{ewp}\ (\mathtt{continue}\ k\ w)\ \langle \Psi' \rangle\ \{Q'\} \\
&\qquad\quad \} \twoheadrightarrow \\
&\qquad\quad \mathtt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle \Psi_2 \rangle\ \{Q_2\})
\end{aligned}
$$

Reasoning about handlers

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

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

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

$$\wedge$$

$$(\forall v\ k.$$

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

$$\triangleright\ \mathit{isDeepHandler}\ \langle\Psi_1\rangle\ \{Q_1\}\ (\boldsymbol{h}\ |\ \boldsymbol{r})\ \langle\Psi'\rangle\ \{Q'\} \twoheadrightarrow$$

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

$$\} \twoheadrightarrow$$

$$\mathrm{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle\Psi_2\rangle\ \{Q_2\})$$

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'.

Application of Hazel

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:

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

Definition of *isIter*

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

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

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
```

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

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
```

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

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

$$\mathit{isSeq}(\mathtt{k}, xs) \triangleq \mathit{isSeq}'(\mathtt{k}, [\,], xs)$$

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

```
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 *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

```
effect Yield : int -> unit
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];
      Cons (x, continue k)
  | ()                 ->
      Empty
```

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

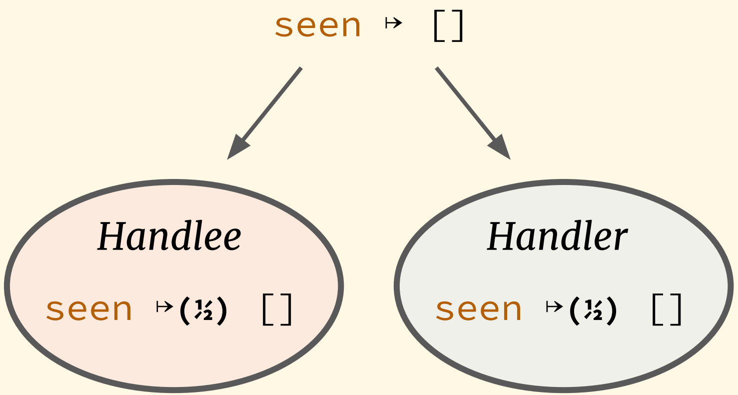
Ghost state is a standard verification technique, usually presented as *history variables*.

Ghost state

```
effect Yield : int -> unit
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];
      Cons (x, continue k)
  | ()                 ->
      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*:

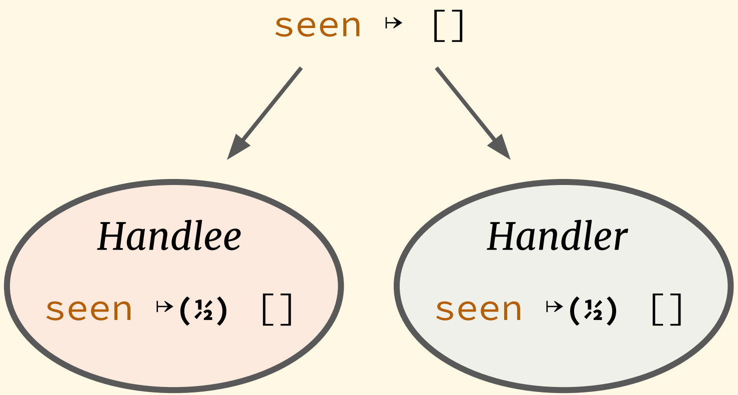
$$\texttt{seen} \mapsto_{(1/2)} us \;\;-\!\!*\;\; \texttt{seen} \mapsto_{(1/2)} vs \;\;-\!\!*\;\; \Rrightarrow\; \texttt{seen} \mapsto (us \mathbin{+\!+} [u]) \;\ast\; us = vs$$

Ghost state

```
effect Yield : int -> unit
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];
      Cons (x, continue k)
  | ()                    ->
      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`

$$\text{(Deep Handler)}\;\frac{\begin{array}{l}\texttt{seen} \mapsto_{(1/2)} [\,] \mathrel{-\!\!*} \\ \texttt{ewp (iter yield)}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \\ \quad \texttt{seen} \mapsto_{(1/2)} \mathit{xs}\}\end{array} \qquad \begin{array}{l}\texttt{seen} \mapsto_{(1/2)} [\,] \mathrel{-\!\!*} \\ \mathit{isDeepHandler} \\ \quad \langle \mathit{YIELD} \rangle\ \{\_.\ \texttt{seen} \mapsto_{(1/2)} \mathit{xs}\} \\ \qquad (\boldsymbol{h} \mid \boldsymbol{r}) \\ \quad \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,[\,],\mathit{xs})\}\end{array}}{\begin{array}{l}\texttt{seen} \mapsto [\,] \mathrel{-\!\!*} \\ \texttt{ewp (match iter yield with} \\ \qquad \texttt{| effect (Yield x) k -> } \boldsymbol{h}\texttt{ x k} \\ \qquad \texttt{|() -> } \boldsymbol{r}\texttt{ ())}\ \ \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,[\,],\mathit{xs})\}\end{array}}$$

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

Verification of `invert`

<u>First proof obligation</u>

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

The first proof obligation follows from the hypothesis $\mathit{isIter}(\texttt{iter}, \mathit{xs})$.

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"*.

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

Verification of `invert`

Second proof obligation

$$\texttt{seen} \mapsto_{(½)} [\,] \;-\!\!*\; \begin{array}{l} \mathit{isDeepHandler}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \texttt{seen} \mapsto_{(½)} \mathit{xs}\} \\ \qquad\qquad (\mathbf{h} \mid \mathbf{r}) \\ \qquad\quad \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,[\,],\mathit{xs})\} \end{array}$$

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

$$H \triangleq \forall \mathit{us}.\ \texttt{seen} \mapsto_{(½)} \mathit{us} \;-\!\!*\; \begin{array}{l} \mathit{isDeepHandler}\ \langle \mathit{YIELD} \rangle\ \{\_.\ \texttt{seen} \mapsto_{(½)} \mathit{xs}\} \\ \qquad\qquad (\mathbf{h} \mid \mathbf{r}) \\ \qquad\quad \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,\mathit{us},\mathit{xs})\} \end{array}$$

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

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

Conclusion

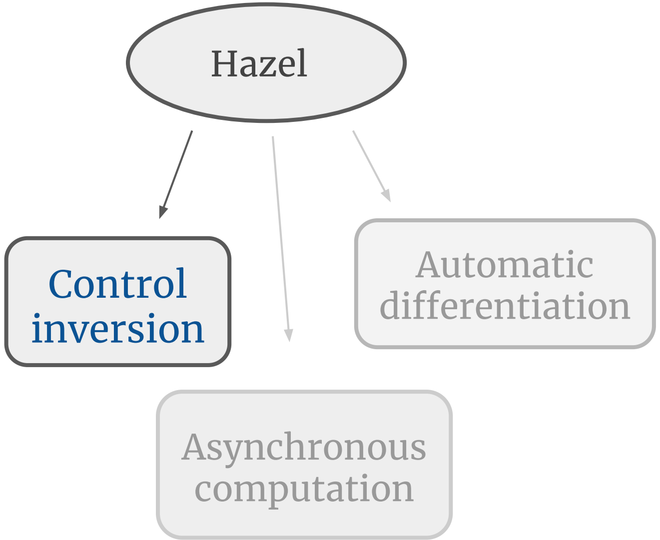
Conclusion

In this talk, I presented *Hazel*
a *Separation Logic* for *effect handlers*.

Hazel preserves *local* reasoning about *state* (*Frame Rule*)
and *context-local* reasoning (*Bind Rule*)

Hazel introduces the notion of *protocols*,
which allows *local* reasoning about *effects*.

Hazel is *successfully* applied to the study of *control inversion*.

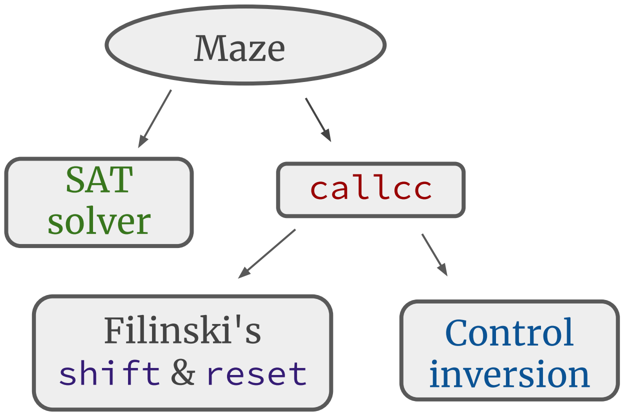


Conclusion

Several contributions have not been discussed today...

Maze, a *Separation Logic* for handlers with *multi-shot continuations*.

Maze is applied to several interesting *case studies*, including `callcc` and Filinski's `shift`/`reset`.

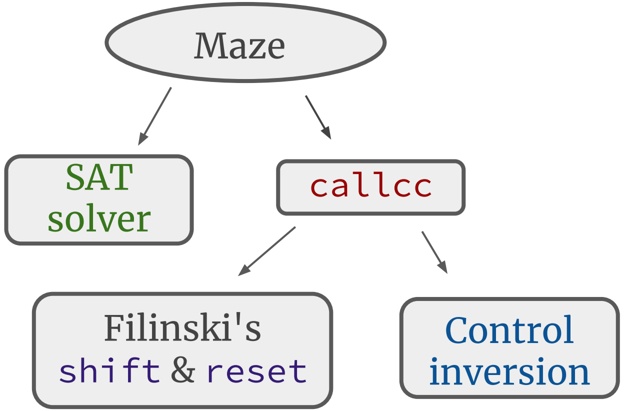


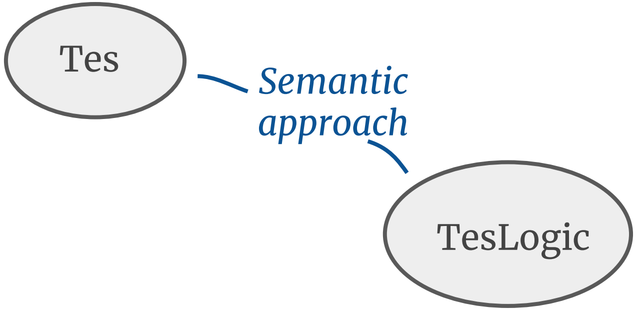
Conclusion

Several contributions have not been discussed today...

Maze, a *Separation Logic* for handlers with *multi-shot continuations*.

Maze is applied to several interesting *case studies*, including `callcc` and Filinski's `shift`/`reset`.





Tes, a *type system* for *effect handlers* and *dynamic effect names*.

Tes's *strong type soundness* follows by the *semantic approach*, which bridges *type systems* to *Separation Logic*.

Questions

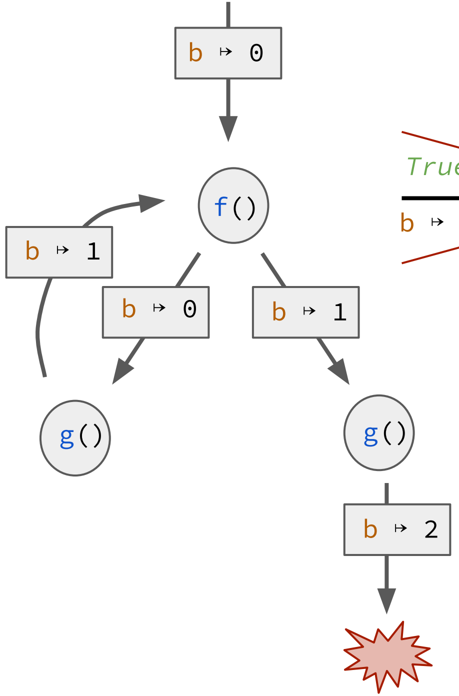
Why multi-shot continuations break the frame rule?

```
effect Escape : unit

let f() = perform Escape

let b = ref 0
let g() = incr b; assert (!b = 1)

let _ =
  match f(); g() with
  | effect Escape k ->
     continue (Obj.clone_continuation k) ();
     continue k () (* Assertion fails! *)
  | () -> ()
```



$$\cancel{\frac{\mathit{True} \;-\!\!*\; \mathrm{ewp}\ f()\ \langle \mathit{ESC} \rangle\ \{\_.\,\mathit{True}\}}{b \mapsto 0 \;-\!\!*\; \mathrm{ewp}\ f()\ \langle \mathit{ESC} \rangle\ \{\_.\,b \mapsto 0\}}}$$

The function `f` *exits twice*:
in *the first time* it terminates, the assertion `b ↦ 0` holds,
but, in *the second time* it terminates, this assertion no longer holds.

Contributions of Tes

Aliasing challenge: effect names may have aliases.

The literature proposes *two solutions* to address the *aliasing challenge*:
(1) *Effect coercions*;
(2) *Dynamic allocation of effect labels* + restriction to *lexically scoped handlers*.

Tes considers the *dynamic allocation of effect labels* as a construct on its own;
a *lexically scoped handler* can be expressed as *derived construct*.

```
effect Not_found : 'a

let find f xs =
  let effect Found : int -> 'a in
  match
    List.iter (fun x ->
      if f x then perform Found x) xs
  with
  | effect (Found x) _ -> x
  | () -> perform Not_found
```

In Tes, the type of `find`
(1) does *not* mention *local effects names*,
(2) includes *non-aliasing assumptions* (`Not_found` ≠ θ).

```
∀θ. (int -{θ}-> bool) ->
       int list -{Not_found.θ}->
         int
```

Summary of Maze

A weakest precondition assertion in Maze assumes the same shape as in Hazel:

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

Moreover, Maze preserves most of Hazel's *reasoning rules* with the exception of

1. The *frame rule*, which is *unsound* in Maze;
2. The *handler rules*,
   which is adapted to allow reasoning about multiple invocations of the continuation; and
3. The rule for *performing effects*,
   which includes a *persistently modality*,
   justifying that the *handlee* can be resumed multiple times.

(*Send/recv*)

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

Reasoning Rules for `callcc`

$$\mathit{persistent}\ (\mathit{isCont}\ \mathtt{k}\ \Phi)$$

$$\frac{\mathit{isCont}\ \mathtt{k}\ \Phi \qquad \Box(\forall \mathtt{w}.\ \Phi'(\mathtt{w}) \mathrel{-\!\!*} \Phi(\mathtt{w}))}{\mathit{isCont}\ \mathtt{k}\ \Phi'}$$

$$\frac{\mathit{isCont}\ \mathtt{k}\ \Phi \mathrel{-\!\!*} \mathtt{wp\ e}\ \{\Phi\}}{\mathtt{wp\ (callcc\ k.\ e)}\ \{\Phi\}}$$

$$\frac{\mathit{isCont}\ \mathtt{k}\ \Phi \qquad \Phi(\mathtt{w})}{\mathtt{wp\ (throw\ k\ w)}\ \{\_.\mathit{False}\}}$$

$$\mathtt{wp\ e}\ \{\Phi\} \triangleq \mathtt{ewp\ e}\ \langle \mathit{CT} \rangle\ \{\Phi\}$$

Reasoning Rules for Filinski's `shift`/`reset`

$$\frac{\texttt{wp e}\ \langle \texttt{Some}\ \Phi \rangle\ \{\Phi\}}{\texttt{wp (reset e)}\ \langle \_ \rangle\ \{\Phi\}}$$

$$\frac{\Box(\forall w.\ \Phi'(w) \twoheadrightarrow \texttt{wp (k w)}\ \langle \_ \rangle\ \{\Phi\}) \twoheadrightarrow \texttt{wp e}\ \langle \texttt{Some}\ \Phi \rangle\ \{\Phi'\}}{\texttt{wp (shift k. e)}\ \langle \texttt{Some}\ \Phi \rangle\ \{\Phi'\}}$$

$$\begin{aligned}
&\texttt{wp e}\ \langle \texttt{Some}\ \Phi \rangle\ \{\Phi'\} \triangleq \\
&\quad \texttt{isMetaCont (Some}\ \Phi\texttt{)} \twoheadrightarrow \texttt{ewp e}\ \langle CT \rangle\ \{y.\ \Phi'(y) * \texttt{isMetaCont (Some}\ \Phi\texttt{)}\}
\end{aligned}$$

$$\begin{aligned}
&\texttt{isMetaCont}\ \mathit{opt} && \triangleq\ \exists k.\ \mathit{mc} \mapsto k * \texttt{inMetaCont}\ \mathit{opt}\ k \\
&\texttt{inMetaCont (Some}\ \Phi\texttt{) k} && \triangleq\ \forall y.\ \Phi(y) \twoheadrightarrow \mathit{mc} \mapsto _ \twoheadrightarrow \texttt{ewp (k y)}\ \langle CT \rangle\ \{_.\mathit{False}\} \\
&\texttt{inMetaCont None}\ _ && \triangleq\ \mathit{True}
\end{aligned}$$

Syntax of protocols

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

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

Remark.

Hazel's *send/recv protocols* are inspired by *Actris's protocols* [Hinrichsen et al, 20], used to describe the interaction between actors in *message-passing concurrency*.

A Hazel *send/recv protocol* is a coinductive Actris protocol defined as the *repetition* of a send/recv pair.

Control inversion using `callcc`

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


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


let invert (iter : iter) : sequence =
  fun () -> callcc kc.
    let r = ref kc in
    let yield u = callcc kp.
      throw !r (Cons (u, fun () ->
        callcc kc. r := kc; throw kp ()))
    in
    iter yield; throw !r Nil
```

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


isSeqCC'(k, us, xs) ≜
     ewp k() ⟨CT⟩ {y. isHeadCC(y, us, xs)}

isHeadCC(y, us, xs) ≜ …
isSeqCC(k, xs) ≜ isSeqCC'(k, [], xs)


∀iter xs.
  isIterCC(iter, xs) −∗
  ewp (invert iter) ⟨CT⟩ {k. isSeqCC(k, xs)}
```