Preuve de Programmes avec Effect Handlers

Paulo Emílio de Vilhena

sous la direction de François Pottier

06/06/2023

cnrs GDR Groupement de recherche
GPL Génie de la programmation et du logiciel

Aperçu

But. Étendre la *logique de séparation* avec support pour les *effect handlers*.

Motivation.

1. **Vérification.** Une telle logique nous permet de *vérifier* qu'un programme est correctement implémenté.

2. **Compréhension.** Réfléchir sur un programme en termes des *spécifications* et *règles de raisonnement* est un outil important pour sa *compréhension*.

3. **Documentation.** La spécification d'un programme sert comme une *documentation* précise de son comportement.

Effect handlers

Les *effect handlers* sont une généralisation des *exception handlers*:

Alors que *lever* une exception *jette* le calcul,

lancer un *effet* suspend le calcul, qui est *réifié* en tant qu'une *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
```

```
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
```

(Exemples écrits en Multicore OCaml 4.12)

Effect handlers

Les *effect handlers* sont une généralisation des *exception handlers*:

Alors que *lever* une exception *jette* le calcul,

lancer un *effet* suspend le calcul, qui est *réifié* en tant qu'une *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
```

```
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 = 0
```

(Exemples écrits en Multicore OCaml 4.12)

Effect handlers

Les *effect handlers* sont une généralisation des *exception handlers*:

Alors que *lever* une exception *jette* le calcul,

lancer un *effet* suspend le calcul, qui est *réifié* en tant qu'une *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
```

(*Exemples écrits en Multicore OCaml 4.12*)

Effect handlers

Il existe *deux* versions des handlers:

- *shallow handlers*, qui attrappent seulement le premier effet; et
- *deep handlers*, qui attrapent tous les effets.

```
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 des effect handlers

Cette capacité de *suspendre* un programme et le *reprendre* plus tard est *très puissante*.

Plusieurs *applications* intéressantes des effect handlers en découlent:

- *Inversion de contrôle.*

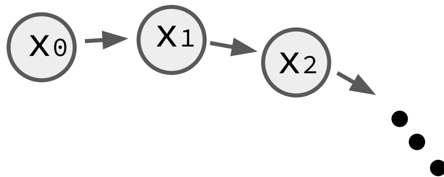
- *Calcul asynchrone.*

Applications des effect handlers

Cette capacité de *suspendre* un programme et le *reprendre* plus tard est *très puissante*.

Plusieurs *applications* intéressantes des effect handlers en découlent:

- *Inversion de contrôle.*



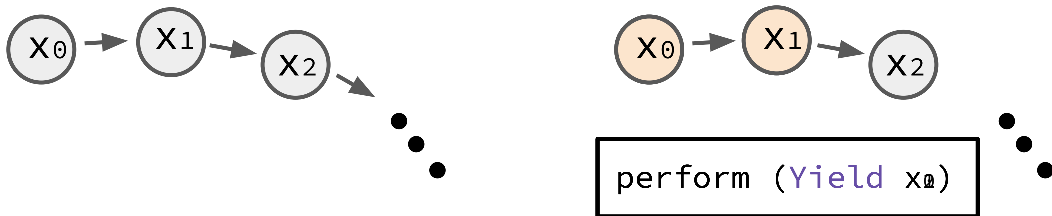
- *Calcul asynchrone.*

Applications des effect handlers

Cette capacité de *suspendre* un programme et le *reprendre* plus tard est *très puissante*.

Plusieurs *applications* intéressantes des effect handlers en découlent:

- *Inversion de contrôle.*



- *Calcul asynchrone.*

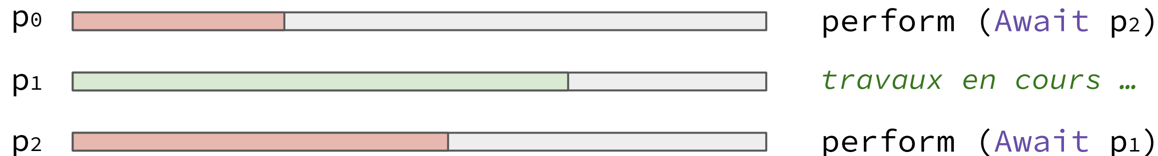
Applications des effect handlers

Cette capacité de *suspendre* un programme et le *reprendre* plus tard est *très puissante*.

Plusieurs *applications* intéressantes des effect handlers en découlent:

- *Inversion de contrôle.*

- *Calcul asynchrone.*

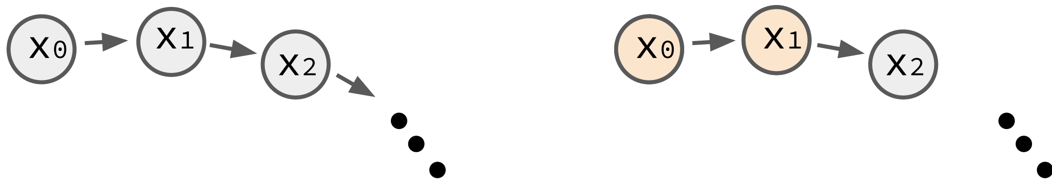


Applications des effect handlers

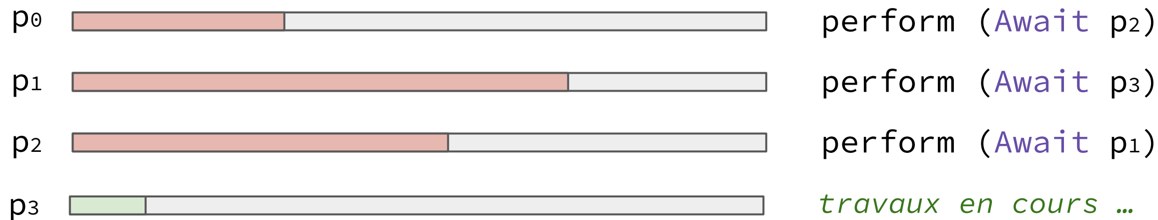
Cette capacité de *suspendre* un programme et le *reprendre* plus tard est *très puissante*.

Plusieurs *applications* intéressantes des effect handlers en découlent:

- *Inversion de contrôle.*

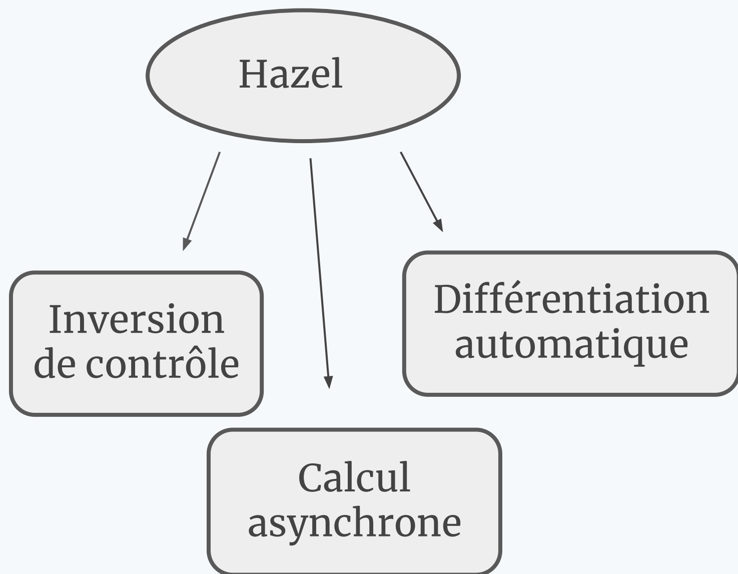


- *Calcul asynchrone.*



Contributions de cette thèse

Cette thèse introduit *Hazel*, une *logique de séparation* pour les effect handlers.



Hazel

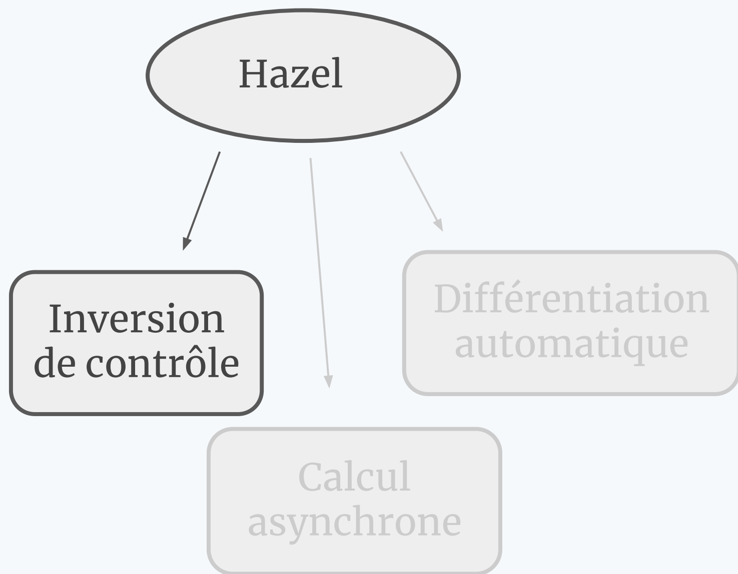
- permet la *spécification* et *vérification* des handlers,
- préserve le raisonnement local à propos de l'*état*,
- permet des nouvelles formes de modularité:
  handlee (*lance* les effets) **VS**
  handler (*attrape* / *"handles"* les effets).

L'*applicabilité* de Hazel est évalué à travers plusieurs *cas d'études*:

- *Inversion de contrôle*
- *Calcul asynchrone*
- *Différentiation automatique*

Contributions de cette thèse

Dans la prochaine partie de cet exposé, je présente *Hazel* et son application à l'*inversion de contrôle*.



Hazel

- permet la *spécification* et *vérification* des handlers,
- préserve le raisonnement modulaire à propos de l'*état*,
- permet des nouvelles formes de modularité:
  handlee (*lance* les effets) **VS**
  handler (*attrape* / "*handles*" les effets).

L'*applicabilité* de Hazel est évalué à travers plusieurs *cas d'études*:

- *Inversion de contrôle*
- *Calcul asynchrone*
- *Différentiation automatique*

Inversion de contrôle

```
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
```

Inversion de contrôle

```
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
```

Une méthode de itération d'ordre supérieur est avide: elle applique une fonction donnée à tous les éléments d'une structure.

Inversion de contrôle

```
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
```

Une séquence paresseuse est une suspension (“thunk”), qui, quand repris, produit soit un marqueur signalant sa fin soit une paire de l'élément tête et la queue.

Inversion de contrôle

```
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
```

La fonction `invert` *transforme une* *méthode d'itération* *vers une* *séquence.*

En gros, `invert` *utilise l'effet* `Yield` *à fin d'interrompre l'itération.*

Spécification de `invert` en Hazel

En Hazel, on peut écrire de façon très concise ce que fait `invert`:

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

- *Précondition* `isIter(iter, xs)`
  dit que `iter` est une *méthode d'itération* pour une structure contenant les éléments `xs`.
- *Postcondition* `isSeq(k, xs)`
  dit que `invert` produit une *séquence* `k` qui produit le même ensemble d'éléments, `xs`.
- *Protocole* ⊥
  dit que `invert` ne lance pas d’effets.

Suite de l'exposé

Après l'introduction de Hazel, on applique cet outil pour étudier `invert` et pour prouver sa spéc.:

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

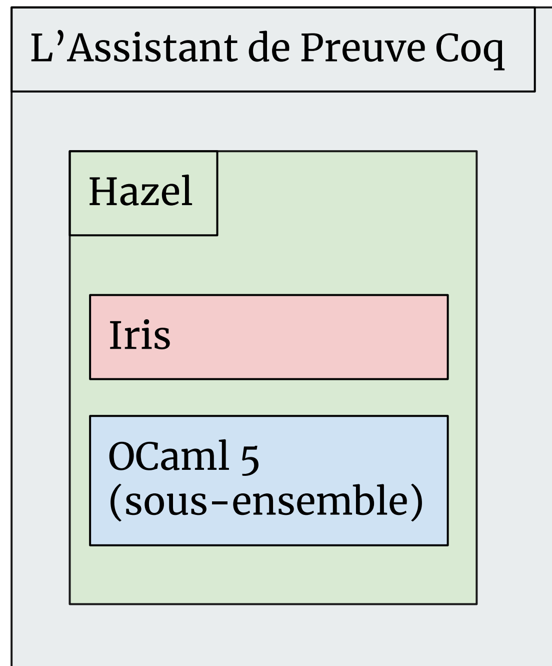
Plus spécifiquement, on va voir

- *La notion de protocoles*
- *La définition de* `isIter(iter, xs)`
- *La définition de* `isSeq(k, xs)`

Hazel

Aperçu du projet Hazel

Hazel est une extension d'*Iris*.



Iris est une logique de séparation moderne:

connecteurs logiques usuels ($\forall$, $\exists$, $\Rightarrow$, $\wedge$, $\vee$),
conjonction séparante ($*$),
baguette magique ($-\!\!*$),
modalité *later* ($\triangleright$, pour les *définitions récursives*),
modalité *persistance* ($\square$, pour décrire les *ressources duplicables*),
modalité *update* ($\Rrightarrow$, pour le support à l'*état fantôme*,
une technique de vérification appliquée à `invert`).

Formalisation de la sémantique opérationnelle d'un sous-ensemble d'*OCaml 5* contenant

(1) *l'état mutable dynamiquement alloué*,
(2) *les effect handlers* (*shallow* et *deep*),
(3) *les noms globaux d'effets*, et
(4) *les continuations one-shot*.

Protocoles

En logique de séparation traditionnelle,
une spécification inclut une *précondition* P et une *postcondition* Q :

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

L'*idée clé* dans Hazel est de généraliser une spécification avec un *protocole* Ψ,
une description des *effets* qu'un programme peut lancer.

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

"Si la précondition P *tient, alors* `e` *peut être exécutée de façon sûre;*
C'est-à-dire, ce programme peut soit
(1) diverger, soit
(2) terminer dans un état où la postcondition Q tient, soit
(3) lancer un effet selon le protocole Ψ*."*

Syntaxe des protocoles

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

- ***Protocole vide*** ⊥
- ***Protocole send/recv*** !x (v) {P}. ?y (w) {Q}
- ***Protocole somme*** Ψ_1 + Ψ_2

Syntaxe des protocoles

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

- ***Protocole vide*** $\bot$

  décrit l'*absence d'effets*.

Exemples.

```
ewp (ref 0) ⟨⊥⟩ {r. r ↦ 0}

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

Syntaxe des protocoles

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

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

  lie une *précondition* P et une *postcondition* Q à un effet,

  suggérant ainsi que *lancer un effet* peut être vu comme *appeller une fonction*.

"Un programme peut lancer un effet u *s'il existe* x *tel que* $u = v$ *et que* P *tienne.*
Pour tout y *, le calcul suspendu peut être repris avec une valeur* w *, à condition que* Q *tienne."*

Syntaxe des protocoles

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

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

Exemples.

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

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

$True \mathrel{-\!\!*} \texttt{ewp}\ (\texttt{perform}\ (\texttt{Abort}\ ()))\ \langle ABORT \rangle\ \{_.\ False\}$

Syntaxe des protocoles

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

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

Exemples.

```
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}
```

Syntaxe des protocoles

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

- ***Protocole somme*** $\Psi_1 + \Psi_2$

  décrit un effet respectant *soit* Ψ_1 *soit* Ψ_2.

Syntaxe des protocoles

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

- ***Protocole somme*** Ψ_1 + Ψ_2

Exemples.

```
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}
```

Règles de raisonnement

Raisonnement sur les effets

(Somme)

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

(Vide)

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

(Send/recv)

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

Raisonnement sur les effets

(Vide)

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

(Somme)

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

(Send/recv)

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

Raisonnement sur les effets

(Somme)

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

(Vide)

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

(Send/recv)

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

Raisonnement sur les effets

(Vide)

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

(Somme)

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

(Send/recv)

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

Raisonnement sur les effets

(Somme)

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

(Vide)

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

"... peut lancer ... u
s'il existe x *tel que* u = v
et que [la précondition] P *tienne ..."*

(Send/recv)

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

Raisonnement sur les effets

(Vide)

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

(Somme)

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

"... pour tout y, le calcul interrompu peut être repris avec... $\texttt{w}$, à condition que [la postcondition] Q tienne."

(Send/recv)

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

Raisonnement local sur l'état

(Règle de l'encadrement)

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

C'est une règle cruciale en *logique de séparation*.

Elle nous permet de raisonner sur des programmes *indépendamment*
à condition qu'ils ne modifient pas les mêmes structures de données.

Hazel *préserve* la *règle d'encadrement*
grâce à la restriction aux *continuations one-shot*.

Raisonnement local sur le contexte d'évaluation

(*Règle de liaison*)

$$\frac{\texttt{ewp e } \langle \Psi \rangle \texttt{ \{y. ewp N[y] } \langle \Psi \rangle \texttt{ \{}Q\texttt{\}\}} \qquad \texttt{N} \text{ est } \textit{neutre}}{\texttt{ewp N[e] } \langle \Psi \rangle \texttt{ \{}Q\texttt{\}}}$$

Un *contexte neutre* ne contient pas de handlers.

Cette règle nous permet de raisonner sur un programme *indépendamment* du contexte sous lequel il sera évalué.

Raisonnement sur le handler

(*Shallow Handler*)

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

Cette règle permet au *handlee* `e` d'être étudié *indépendamment* du *handler* qui monitore son exécution.

Intuitivement, le *protocole* Ψ_1 est une frontière d'abstraction entre *handlee* et *handler*: *lancer des effets* est similaire à *envoyer des requêtes à un serveur,* dont *l'interface* Ψ_1 le handler doit *implémenter*.

Raisonnement sur le handler

Le *jugement shallow-handler* `isShallowHandler` comprend
les spécifications de la *branche return* et de la *branche effet*:

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

$$(\forall y.\ Q_1(y) -\!\!*\ \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle \Psi_2 \rangle\ \{Q_2\}) \qquad \textit{(Branche return)}$$

$$\wedge$$

$$(\forall v\ k.\ \mathtt{ewp}\ (\mathtt{perform}\ v)\ \langle \Psi_1 \rangle\ \{w.\ \mathtt{ewp}\ (\mathtt{continue}\ k\ w)\ \langle \Psi_1 \rangle\ \{Q_1\}\} -\!\!*\ \mathtt{ewp}\ (\boldsymbol{h}\ v\ k)\ \langle \Psi_2 \rangle\ \{Q_2\}) \qquad \textit{(Branche effet)}$$

Raisonnement sur le handler

Le *jugement shallow-handler* `isShallowHandler` comprend les spécifications de la *branche return* et de la *branche effet*:

```
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})
```

La *branche return* peut supposer que y satisfait la *postcondition* du handlee Q_1.

Raisonnement sur le handler

Le *jugement shallow-handler* `isShallowHandler` comprend les spécifications de la *branche return* et de la *branche effet*:

```
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}
```

La *branche effet* peut supposer que `v` a été lancé sous un contexte (représenté par) `k` selon le *protocole* Ψ_1.

Raisonnement sur le handler

Le *jugement shallow-handler* `isShallowHandler` comprend
les spécifications de la *branche return* et de la *branche effet*:

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

$$(\forall y.\ Q_1(y) \mathrel{-\!\!*} \texttt{ewp}\ (\boldsymbol{r}\ y)\ \langle \Psi_2 \rangle\ \{Q_2\})$$

$$\wedge$$

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

On identifie la *permission* à reprendre la continuation.

La continuation `k` peut être reprise avec une valeur `w`,
à condition que `w` soit permis par Ψ_1.

On peut supposer que l'expression `continue k w`
lance des effets selon le protocole Ψ_1
et *termine* selon la postcondition Q_1.

Raisonnement sur le handler

(*Deep Handler*)

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

La règle de raisonnement pour les *deep handlers* est similaire à la règle pour les *shallow handlers*, La différence est cachée dans la définition du *jugement deep-handler* `isDeepHandler`.

Raisonnement sur le handler

Le *jugement deep-handler* `isDeepHandler` est une définition récursive, ce qui est en accord avec le comportement récursif des deep handlers.

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

Raisonnement sur le handler

Le *jugement deep-handler* `isDeepHandler` est une définition récursive,
ce qui est en accord avec le comportement récursif des deep handlers.

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

$$(\forall y.\ Q_1(y) \mathrel{-\!\!*} \mathtt{ewp}\ (\boldsymbol{r}\ y)\ \langle \Psi_2 \rangle\ \{Q_2\})$$

$$\land$$

$$(\forall v\ k.$$

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

$$\triangleright\ \mathtt{isDeepHandler}\ \langle \Psi_1 \rangle\ \{Q_1\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle \Psi' \rangle\ \{Q'\} \mathrel{-\!\!*}$$

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

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

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

Pour raisonner sur un appel à la continuation,
on doit *rétablir* le jugement handler,
car le handler est *réinstallé*.

Cette nouvelle instance du handler peut
se comporter selon un *protocole différent* Ψ'
et selon une *postcondition différente* Q'.

Application de Hazel

Spécification de `invert`

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

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

val invert : iter -> sequence
```

On souhaite prouver que `invert` satisfait la spécification suivante:

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

Définition de *isIter*

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

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

Le *prédicat abstrait* I est l'*invariant de boucle*:

"Si `f` *peut avancer d'un pas, alors* `iter` *peut avancer de* `xs` *pas."*

Définition de *isIter*

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

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

Le *prédicat abstrait* I est l'*invariant de boucle*.

Le *protocole abstrait* Ψ capture l'idée que `iter` soit *polymorphe en effets*:

(1) `iter` *ne lance pas* des effets, et

(2) `iter` *ne capture pas* les effets lancés par `f` .

Définition de *isSeq*

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

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

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

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

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

$$\quad \text{end}$$

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

Le protocole $\bot$ indique que la séquence *ne lance pas d'effets*.

Comme la définition de *isSeq'* n'inclut pas une *modalité persistance*,
la séquence k *n'est pas* duplicable; elle peut être appelée *au plus une fois*.

Idées clés

```
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
```

On a introduit les définitions, maintenant on va étudier les *idées clés* de la preuve:

1. L'introduction d'un morceau d'*état fantôme* pour garder trace des éléments visités.
2. L'introduction d'un protocole décrivant l'effet `Yield`.

État fantôme

```
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
```

La cellule mémoire `seen` fait partie de l'*état fantôme*, une *extension fictionnelle du tas*.

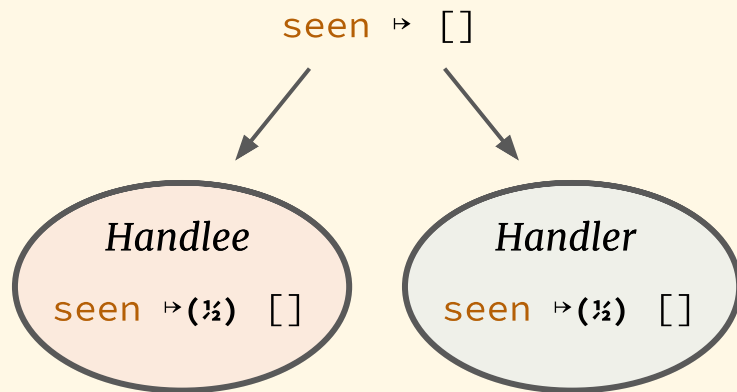
État fantôme est une technique de vérification usuelle, connue aussi sous le nom des “*history variables*”.

État fantôme

```
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
```

La *possession* de la *location fantôme* `seen` est divisée entre *handlee* et *handler*:



Pour modifier `seen`, il faut la *possession complète*, ce qui peut être récupéré d'après les *deux moitiés*:

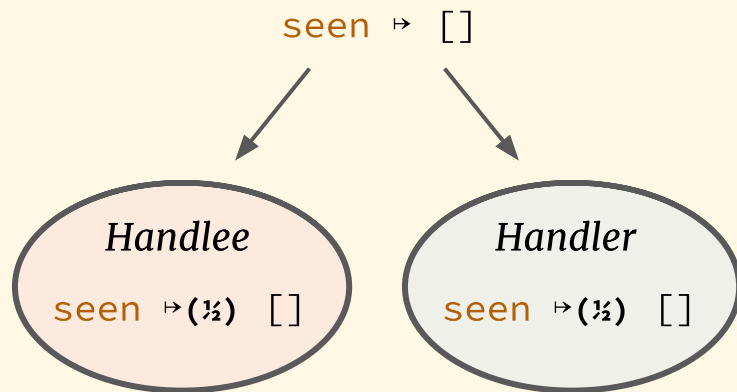
$$\texttt{seen} \mapsto^{(\frac{1}{2})} us \ \twoheadrightarrow^{*} \ \texttt{seen} \mapsto^{(\frac{1}{2})} vs \ \twoheadrightarrow^{*} \ \Rrightarrow \ \texttt{seen} \mapsto (us \mathbin{++} [u]) \ast us = vs$$

État fantôme

```
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
```

La *possession* de la *location fantôme* seen est divisée entre *handlee* et *handler*:



"Du point de vue du handlee, l'effet Yield *u ajoute l'élément u à* seen*."*

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

Vérification de `invert`

$$\frac{\texttt{seen} \mapsto^{(½)} [\,] \;-\!\!*\; \texttt{ewp (iter yield)} \;\langle \mathit{YIELD} \rangle\; \{_.\; \texttt{seen} \mapsto^{(½)} \mathit{xs}\} \qquad \texttt{seen} \mapsto^{(½)} [\,] \;-\!\!*\; \mathit{isDeepHandler}\; \langle \mathit{YIELD} \rangle\; \{_.\; \texttt{seen} \mapsto^{(½)} \mathit{xs}\}\; (\boldsymbol{h} \mid \boldsymbol{r})\; \langle \bot \rangle\; \{y.\; \mathit{isHead}(y, [\,], \mathit{xs})\}}{\texttt{seen} \mapsto [\,] \;-\!\!*\; \texttt{ewp (match iter yield with | effect (Yield x) k -> } \boldsymbol{h}\texttt{ x k | () -> } \boldsymbol{r}\texttt{ ())}\; \langle \bot \rangle\; \{y.\; \mathit{isHead}(y, [\,], \mathit{xs})\}}\;(\textit{Deep Handler})$$

Après l'allocation de `seen`, il suit *le pas de raisonnement le plus important*:
l'application de la *Règle Deep Handler*.

Vérification de `invert`

<u>Première obligation de preuve</u>

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

La première obligation de preuve suit d'après l'hypothèse $\mathit{isIter}(\texttt{iter}, \mathit{xs})$.

En effet, il suffit de

(1) instancier l'invariant de boucle $I(\mathit{us})$ avec $\texttt{seen} \mapsto_{(½)} \mathit{us}$,

(2) instancier le protocole abstrait Ψ avec YIELD, et

(2) prouver que la fonction `yield` *"avance l'invariant d'un pas".*

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

Vérification de `invert`

Deuxième obligation de preuve

$$\texttt{seen} \mapsto_{(\frac{1}{2})} [\,] \;-\!\!*\; \mathit{isDeepHandler}\ \langle \mathit{YIELD} \rangle\ \{_.\ \texttt{seen} \mapsto_{(\frac{1}{2})} xs\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,[\,],xs)\}$$

D'abord, on généralise cette assertion pour raisonner à propos d'un état arbitraire de `seen`:

$$H \triangleq \forall us.\ \texttt{seen} \mapsto_{(\frac{1}{2})} us \;-\!\!*\; \mathit{isDeepHandler}\ \langle \mathit{YIELD} \rangle\ \{_.\ \texttt{seen} \mapsto_{(\frac{1}{2})} xs\}\ (\boldsymbol{h} \mid \boldsymbol{r})\ \langle \bot \rangle\ \{y.\ \mathit{isHead}(y,us,xs)\}$$

Ensuite, on applique l'induction de Löb (car un deep handler est récursivement défini):

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

Conclusion

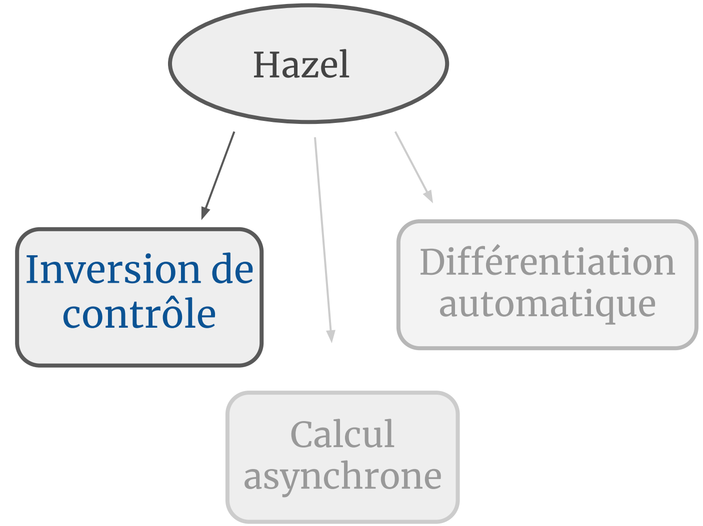
Conclusion

Dans cet exposé, j’ai présenté *Hazel*
une *logique de séparation* pour les *effect handlers*.

Hazel préserve le raisonnement *local* sur
l'*état* (*Règle de l’encadrement*) et le raisonnement
local sur le contexte d'évaluation (*Règle de liaison*)

Hazel introduit la notion de *protocoles*,
qui permet le raisonnement *local* a propos des *effets*.

Hazel est appliqué avec *succès* à l'étude de *l’inversion de contrôle*.

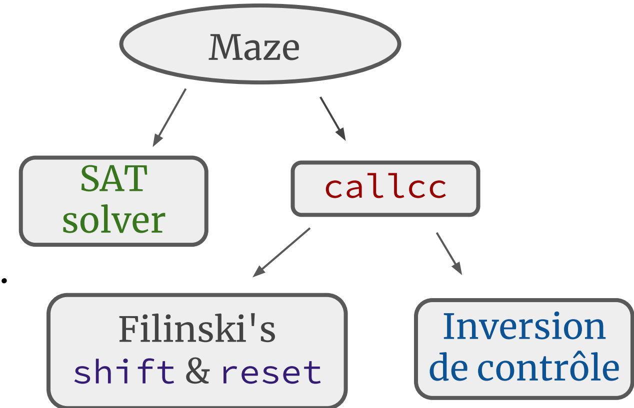


Conclusion

Quelques contributions ont été omises...

Maze, une *logique de séparation*
pour les handlers avec des *continuations multi-shot*.

Maze est appliqué à plusieurs cas d'études intéressants,
y compris `callcc` et l'encodage de Filinski de `shift`/`reset`.

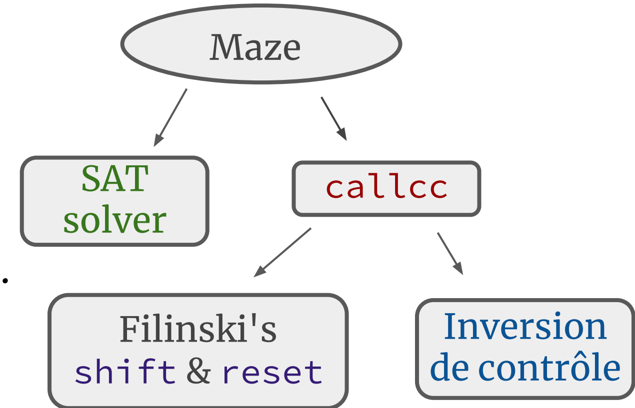


Conclusion

Quelques contributions ont été omises...

Maze, une *logique de séparation*
pour les handlers avec des *continuations multi-shot*.

Maze est appliqué à plusieurs cas d'études intéressants,
y compris `callcc` et l'encodage de Filinski de `shift`/`reset`.



Tes

Approche sémantique

TesLogic

Tes, un *système de types* pour les *effect handlers*
et les *noms locaux d'effets*.

Le résultat de "*sûreté forte*" de Tes suit par l'approche sémantique,
capable de lier les *systèmes de types* à la *logique de séparation*.

Questions

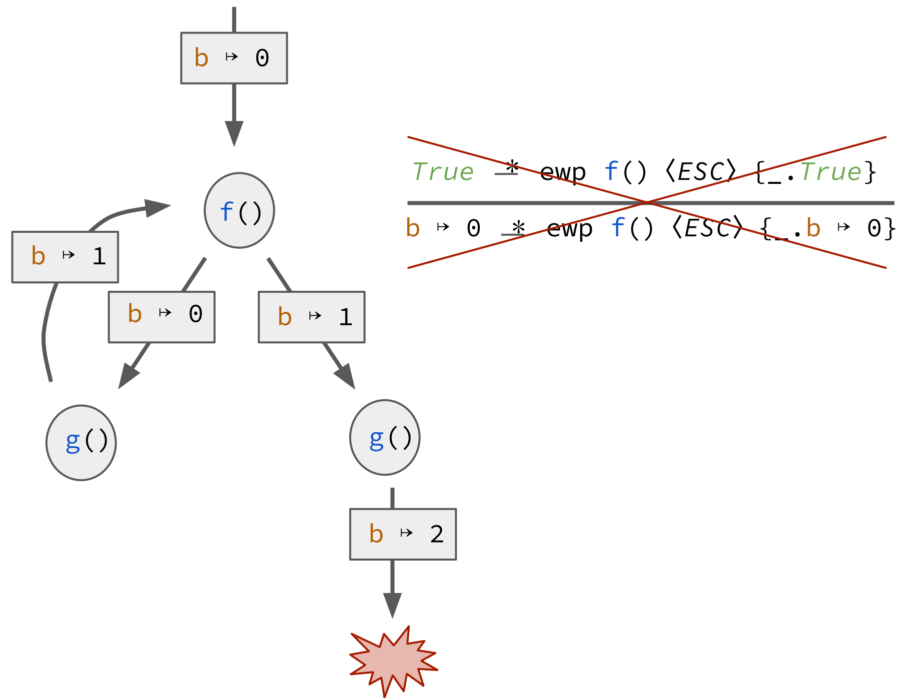
Why multi-shot continuations break the frame rule?

```
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! *)
  | () -> ()
```



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:

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

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*)

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

Reasoning Rules for `callcc`

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

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

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

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

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

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

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

$$\frac{\Box(\forall w.\ \Phi'(w) \mathrel{-\!\!*} \texttt{wp (k w)}\ \langle_\rangle\ \{\Phi\}) \mathrel{-\!\!*} \texttt{wp e}\ \langle\texttt{Some}\ \Phi\rangle\ \{\Phi'\}}{\texttt{wp (shift k. e)}\ \langle\texttt{Some}\ \Phi\rangle\ \{\Phi'\}}$$

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

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

Syntax of protocols

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

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

Proof of Soundness

Our proof of soundness follows the *semantic approach*, which consists of three steps:

1. Translate typing judgments as *specifications* written in a *certain program logic*.
   (In our case, we choose *TesLogic*, a *Separation Logic* with support for *handlers*.)

2. Prove that, if a *typing judgment* is *derivable*, then its *translation holds*.

3. Show that the translation implies the *system's desired guarantees*.

Pictorially,