Structures de contrôle, huitième cours

Logiques de programmes pour le contrôle et les effets

Xavier Leroy

2024-03-14

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Vérification déductive et logique de Hoare

La vérification déductive de programmes

Annoter les programmes par des assertions logiques :

- préconditions : propriétés attendues des entrées;
- postconditions : garanties fournies sur les sorties;
- invariants : associés aux boucles, aux objets, etc.

Exemple (Spécification ACSL d’une fonction C)

```
/*@
  requires \valid(a+(0..n-1));
  assigns  a[0..n-1];
  ensures  \forall integer i; 0 <= i < n ==> a[i] == 0;
*/
void set_to_0(int* a, size_t n)
```

La vérification déductive de programmes

Annoter les programmes par des assertions logiques.

Vérifier la cohérence de ces annotations :

$$\text{préconditions} \Rightarrow \text{invariants} \Rightarrow \text{postconditions}$$

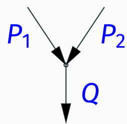
le long de tous les chemins d'exécution possibles dans le programme.

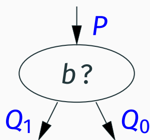
L'approche de Floyd

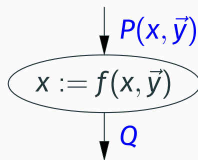
(Alan Turing, *Checking a large routine*, 1949.)
(Robert W. Floyd, *Assigning meanings to programs*, 1967.)

Un graphe de flux de contrôle (organigramme) dont les arcs sont annotés par des assertions.

Vérifier la cohérence des annotations à chaque nœud :







$P_1 \vee P_2 \Rightarrow Q$

$P \wedge b \Rightarrow Q_1$
$P \wedge \neg b \Rightarrow Q_0$

$\big(\exists x_0, x = f(x_0, \vec{y}) \wedge P(x_0, \vec{y})\big) \Rightarrow Q$

L'approche de Hoare

(C. A. R. Hoare, *An axiomatic basis for computer programming*, CACM 12, 1969.)

Une logique de programmes :

Des axiomes et des règles de déduction
permettant d'établir des propriétés de toutes les exécutions
des commandes d'un langage impératif à contrôle structuré.

Un lien fort avec les structures de contrôle
et la programmation structurée :

La forme de la vérification suit la structure du programme.
Les axiomes et les règles suivent les structures de contrôle du langage.

Les triplets de Hoare

$$\{ P \} \ c \ \{ Q \}$$

c : commande d'un langage impératif structuré (Algol, ...)

P, Q : assertions logiques portant sur les variables du programme.

P : précondition, supposée vraie ≪avant≫ l'exécution de c

Q : postcondition, garantie vraie ≪après≫ l'exécution de c

Les triplets de Hoare

Logique de Hoare ≪faible≫ : (correction partielle)

$\{ P \}\ c\ \{ Q \}$ si P est vraie ≪avant≫ et si c termine,
alors Q est vraie ≪après≫

Logique de Hoare ≪forte≫ : (correction totale)

$[P]\ c\ [Q]$ si P est vraie ≪avant≫,
alors c termine et Q est vraie ≪après≫

Les règles de la logique de Hoare faible

Contrôle structuré :

$$\frac{\{\,P\,\}\;c_1\;\{\,Q\,\}\qquad\{\,Q\,\}\;c_2\;\{\,R\,\}}{\{\,P\,\}\;c_1;c_2\;\{\,R\,\}}$$

$$\frac{\{\,P\wedge b\,\}\;c_1\;\{\,Q\,\}\qquad\{\,P\wedge\neg b\,\}\;c_2\;\{\,Q\,\}}{\{\,P\,\}\;\texttt{if}\;b\;\texttt{then}\;c_1\;\texttt{else}\;c_2\;\{\,Q\,\}}$$

$$\frac{\{\,P\wedge b\,\}\;c\;\{\,P\,\}}{\{\,P\,\}\;\texttt{while}\;b\;\texttt{do}\;c\;\{\,P\wedge\neg b\,\}}$$

Les règles de la logique de Hoare

Commande vide :

$$\{\, P \,\}\ \mathtt{skip}\ \{\, P \,\}$$

Affectation :

$$\{\, Q[x \leftarrow e] \,\}\ x := e\ \{\, Q \,\}$$

Conséquence :

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}$$

Exemple de vérification : la division euclidienne

$$\{0 \leq a\} \Rightarrow \{a = b \cdot 0 + a \ \wedge \ 0 \leq a\}$$

$\texttt{r} := \texttt{a};$

$$\{a = b \cdot 0 + r \ \wedge \ 0 \leq r\}$$

$\texttt{q} := 0;$

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r\}$$

$\texttt{while r} \geq \texttt{b do}$

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r \ \wedge \ r \geq b\} \Rightarrow$$
$$\{a = b \cdot (q + 1) + (r - b) \ \wedge \ 0 \leq r - b\}$$

$\quad \texttt{r} := \texttt{r} - \texttt{b};$

$$\{a = b \cdot (q + 1) + r \ \wedge \ 0 \leq r\}$$

$\quad \texttt{q} := \texttt{q} + 1$

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r\}$$

$\texttt{done}$

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r \ \wedge \ r < b\} \Rightarrow$$
$$\{q = a/b \ \wedge \ r = a \bmod b\}$$

Extensions de la logique de Hoare à diverses structures de contrôle

D'autres formes de boucles

La boucle `do...while` avec test d'arrêt à la fin (C, Java) :

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad Q \land b \Rightarrow P}{\{\,P\,\}\ \texttt{do}\ c\ \texttt{while}\ b\ \{\,Q \land \neg b\,\}}$$

Une boucle avec test d'arrêt au milieu (Ada) :

$$\frac{\{\,P\,\}\ c_1\ \{\,Q\,\} \qquad \{\,Q \land \neg b\,\}\ c_2\ \{\,P\,\}}{\{\,P\,\}\ \texttt{loop}\ c_1;\ \texttt{exit when}\ b;\ c_2\ \texttt{end}\ \{\,Q \land b\,\}}$$

Une boucle comptée `for` :

$$\frac{[\,P \land i \leq h\,]\ c\ [\,P[i \leftarrow i+1]\,] \qquad i, h \text{ non affectées dans } c}{[\,P[i \leftarrow \ell]\,]\ \texttt{for}\ \ i = \ell\ \ \texttt{to}\ \ h\ \ \texttt{do}\ \ c\ [\,P \land i > h\,]}$$

Non-déterminisme

Tirer un nombre entre 0 et $N-1$:

$$\{\, \forall i \in [0, N-1],\ Q[\mathrm{x} \leftarrow i\,] \,\}\ x := \mathtt{choose}(N)\ \{\, Q \,\}$$

La «conditionnelle gardées» de Dijkstra :
exécute un des c_i pour lequel b_i est vrai.

$$\frac{\{\, P \wedge b_i \,\}\ c_i\ \{\, Q \,\} \quad \text{pour } i = 1, \ldots, n}{\{\, P \wedge (b_1 \vee \cdots \vee b_n) \,\}\ \mathtt{if}\ \ b_1 \rightarrow c_1 \,[\!]\, \cdots \,[\!]\, b_n \rightarrow c_n\ \ \mathtt{fi}\ \{\, Q \,\}}$$

Une logique de Hoare pour le ≪goto≫ ?

Areas which do present real difficulty are labels and jumps, pointers, and name parameters. Proofs of programs which made use of these features are likely to be elaborate, and it is not surprising that this should be reflected in the complexity of the underlying axioms.

(C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969)

Une logique de Hoare pour le ≪goto≫ ?

On considère le `goto` d'Algol 60 : une étiquette L a pour portée le bloc où elle est définie $\Rightarrow$ pas de saut vers l'intérieur d'un bloc.

```
begin
    ...
    goto L
    ...
 L:
    ...
    begin ...     goto L      ... end;
    ...
end
```

Idée : chaque étiquette L a une précondition R, qui est celle de la commande qui suit. Chaque `goto` L a la même précondition R et une postcondition quelconque.

Une logique de Hoare pour le «goto» ?

On considère le `goto` d'Algol 60 : une étiquette L a pour portée le bloc où elle est définie $\Rightarrow$ pas de saut vers l'intérieur d'un bloc.

```
begin
    ...
{R} goto L {Q1}
    ...
 L: {R}
    ...
    begin ... {R} goto L {Q2} ... end;
    ...
end
```

Idée : chaque étiquette L a une précondition R, qui est celle de la commande qui suit. Chaque `goto` L a la même précondition R et une postcondition quelconque.

La règle de Clint et Hoare pour le «goto»

(M. Clint, C. A. R. Hoare, *Program proving : jumps and functions*, Acta Informatica 1, 1971.)

$$\frac{\{\,R\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\} \vdash \{\,P\,\}\ c_1\ \{\,R\,\} \qquad \{\,R\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\} \vdash \{\,R\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{begin}\ c_1;\ L : c_2\ \texttt{end}\ \{\,Q\,\}}$$

$X \vdash Y$ se lit comme une dérivation hypothétique en déduction naturelle : «en supposant X on peut dériver Y».

De l'hypothèse $\{\,R\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\}$ on peut dériver $\{\,R\,\}\ \texttt{goto}\ L\ \{\,Q\,\}$ pour n'importe quel Q avec la règle de conséquence.

Problème avec la règle de Clint et Hoare

(M. J. O'Donnell, *A critique of the foundations of Hoare style programming logics*, CACM 25, 1982.)

En cas de blocs imbriqués

$$\texttt{begin} \ldots \texttt{begin} \ldots L : \ldots \texttt{end} \ldots L : \ldots \texttt{end}$$

«la» précondition associée à L est ambiguë :

$$\{ R_1 \} \text{ goto } L \{ \texttt{false} \} \vdash (\{ R_2 \} \text{ goto } L \{ \texttt{false} \} \vdash X)$$

De plus, l'interprétation logique de $X \vdash Y$ est délicate. Si on lit comme «il existe un modèle où X implique Y», on peut prendre $X = Y =$ faux, et déduire

$$\frac{\{ \texttt{false} \} \text{ goto } L \{ \texttt{false} \} \implies \{ \texttt{true} \} \text{ goto } L \{ \texttt{false} \}}{✘ \ \{ \texttt{true} \} \text{ begin goto } L;\ L : \text{skip end} \{ \texttt{false} \}}$$

L’approche de Arbib-Alagic-de Bruin

(M. Arbib, S. Alagić, *Proof rules for gotos*, Acta Informatica 11, 1979.
A. de Bruin, *Goto statements : semantics and deduction systems*, Acta Informatica 15, 1981.)

Idée : le `goto` est un autre moyen de ≪sortir≫ de l’exécution d’une commande c, en plus de la terminaison normale.
Il faut donc refléter le `goto` dans une postcondition supplémentaire J :

$$\{ P \}\ c\ \{ Q \}\ \{ J \}$$

J est une fonction étiquette $\mapsto$ assertion. Elle peut être affaiblie comme la postcondition Q normale :

$$\frac{P' \Rightarrow P \qquad \{ P \}\ c\ \{ Q \}\ \{ J \} \qquad Q \Rightarrow Q' \qquad \forall L,\ J(L) \Rightarrow J'(L)}{\{ P' \}\ c\ \{ Q' \}\ \{ J' \}}$$

Les règles de Arbib-Alagic-de Bruin

La postcondition J est fausse pour les commandes qui terminent toujours normalement :

$$\{\, Q[x \leftarrow e] \,\}\ x := e\ \{\, Q \,\}\ \{\, \lambda L.\, \texttt{false} \,\}$$

J est partagée entre les sous-commandes d'une séquence ou d'une conditionnelle :

$$\frac{\{\, P \,\}\ c_1\ \{\, R \,\}\ \{\, J \,\} \qquad \{\, R \,\}\ c_2\ \{\, Q \,\}\ \{\, J \,\}}{\{\, P \,\}\ c_1; c_2\ \{\, Q \,\}\ \{\, J \,\}}$$

$$\frac{\{\, P \wedge b \,\}\ c_1\ \{\, Q \,\}\ \{\, J \,\} \qquad \{\, P \wedge \neg b \,\}\ c_2\ \{\, Q \,\}\ \{\, J \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\, Q \,\}\ \{\, J \,\}}$$

Les règles de Arbib-Alagic-de Bruin

$\texttt{goto}\ L$ a toutes ses postconditions fausses sauf $J(L)$ qui est sa précondition P :

$$\{\,P\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\}\ \{\,\lambda L'.\ \texttt{if}\ L' = L\ \texttt{then}\ P\ \texttt{else}\ \texttt{false}\,\}$$

Dans un bloc qui définit L avec précondition R, toutes les sorties sur $\texttt{goto}\ L$ doivent satisfaire R :

$$\frac{\{\,P\,\}\ c_1\ \{\,R\,\}\ \{\,J[L \leftarrow R]\,\} \qquad \{\,R\,\}\ c_2\ \{\,Q\,\}\ \{\,J[L \leftarrow R]\,\}}{\{\,P\,\}\ \texttt{begin}\ c_1;\ L : c_2\ \texttt{end}\ \{\,Q\,\}\ \{\,J\,\}}$$

Sorties prématurées de boucles

Des constructions comme `break` (sortie prématurée de boucle) peuvent aussi être traitées par une postcondition spéciale :

$$\{\,P\,\}\ c\ \{\,Q\,\}\ \{\,B\,\} \qquad (B = \text{précondition du } \texttt{break})$$

Quelques règles :

$$\{\,P\,\}\ \texttt{break}\ \{\,\texttt{false}\,\}\ \{\,P\,\}$$

$$\frac{\{\,P \wedge b\,\}\ c\ \{\,P\,\}\ \{\,Q\,\} \qquad P \wedge \neg b \Rightarrow Q}{\{\,P\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\,Q\,\}\ \{\,B\,\}}$$

$$\frac{\{\,P \wedge b\,\}\ c\ \{\,P\,\}\ \{\,Q\,\} \qquad \{\,P \wedge \neg b\,\}\ c'\ \{\,Q\,\}\ \{\,B\,\}}{\{\,P\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{else}\ c'\ \{\,Q\,\}\ \{\,B\,\}}$$

Un traitement unifié des sorties multiples

Au lieu d'avoir une postcondition pour chaque manière de sortir d'une commande, on peut faire de la postcondition une fonction

$$Q : \text{type de sortie} \mapsto \text{assertion}$$

Les types de sortie T sont, par exemple,

$T ::= \texttt{norm}$	terminaison normale
$\mid \texttt{break} \mid \texttt{continue}$	sorties de boucles
$\mid \texttt{break}(n) \mid \texttt{continue}(n)$	sorties multi-niveaux
$\mid \texttt{return}(v)$	retour de fonction
$\mid \texttt{goto}(L)$	branchement
$\mid \texttt{exn}(E)$	levée d'exception

Un traitement unifié des sorties multiples

Les règles pour les commandes qui font sortir ont toutes la même forme :

$$\{ P \} \; \texttt{skip} \; \{ [\texttt{norm} \mapsto P] \}$$
$$\{ P \} \; \texttt{break} \; \{ [\texttt{break}(1) \mapsto P] \}$$
$$\{ P \} \; \texttt{break} \; n \; \{ [\texttt{break}(n) \mapsto P] \}$$
$$\{ P \} \; \texttt{goto} \; L \; \{ [\texttt{goto}(L) \mapsto P] \}$$
$$\{ P \} \; \texttt{raise} \; E \; \{ [\texttt{exn}(E) \mapsto P] \}$$

On note $[T \mapsto P] \stackrel{def}{=} \lambda T'. \; \texttt{if} \; T' = T \; \texttt{then} \; P \; \texttt{else} \; \texttt{false}$.

Un traitement unifié des sorties multiples

La séquence «gère» la sortie normale :

$$\frac{\{\,P\,\}\ c_1\ \{\,Q[\texttt{norm} \leftarrow R]\,\} \qquad \{\,R\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ c_1; c_2\ \{\,Q\,\}}$$

La boucle «gère» aussi les sorties `break` et `continue` :

$$Q' = Q\left[\begin{array}{l}\texttt{norm} \leftarrow P;\\ \texttt{break}(1) \leftarrow Q(\texttt{norm});\\ \texttt{break}(n+1) \leftarrow Q(\texttt{break}(n))\\ \texttt{continue}(1) \leftarrow P;\\ \texttt{continue}(n+1) \leftarrow Q(\texttt{continue}(n))\end{array}\right]$$

$$\frac{\{\,P \wedge b\,\}\ c\ \{\,Q'\,\} \qquad P \wedge \neg b \Rightarrow Q(\texttt{norm})}{\{\,P\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\,Q\,\}}$$

Un traitement unifié des sorties multiples

La déclaration de l'étiquette L «gère» la sortie $\texttt{goto}(L)$:

$$\frac{\{\,P\,\}\ c_1\ \{\,Q[\texttt{norm} \leftarrow R, \texttt{goto}(L) \leftarrow R]\,\} \qquad \{\,R\,\}\ c_2\ \{\,Q[\texttt{goto}(L) \leftarrow R]\,\}}{\{\,P\,\}\ \texttt{begin}\ c_1;\ L : c_2\ \texttt{end}\ \{\,Q\,\}}$$

Les gestionnaires d'exceptions «gèrent» les sorties $\texttt{exn}(E)$:

$$\frac{\{\,P\,\}\ c_1\ \{\,Q[\texttt{exn}(E) \leftarrow R]\,\} \qquad \{\,R\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{try}\ c_1\ \texttt{catch}\ E \rightarrow c_2\ \{\,Q\,\}}$$

Les coroutines

(M. Clint, *Program proving : coroutines*, Acta Informatica 2, 1973.)

Un modèle simple de coroutines asymétrique :

$$\texttt{coroutine}\ p = c_1\ \texttt{in}\ c_2$$

Quand le consommateur c_2 fait `call` p, l'exécution de c_1 démarre ou reprend juste après le dernier `yield` p.

Quand le générateur c_1 fait `yield` p, l'exécution de c_2 reprend juste après le dernier `call` p.

Le calcul termine dès que c_1 ou c_2 termine.

La communication de valeurs se fait par variables partagées.

Un exemple de coroutine

```
var obs: int, c: int = 0, h: array [0..N-1] of int = { 0, ... }
coroutine p =
    begin
        while c < N do
            h[obs] := h[obs] + 1; c := c + 1;
            yield p
        done
    end
in
    ... obs = 12; call p; ...
    ... obs = 41; call p; ...
```

La coroutine tient à jour l'histogramme `h` des valeurs observées `obs`, et s'arrête dès que `N` valeurs ont été observées.

Le client appelle `p` sur diverses valeurs `obs`.

La règle de Clint pour les coroutines

Deux assertions associées à la coroutine p :

- A_p : la pré de `call` p, et donc aussi la post de `yield` p ;
- B_p : la pré de `yield` p et donc aussi la post de `call` p.

La règle de Clint :

$$\frac{\{\,B_p\,\}\ \texttt{yield}\ p\ \{\,A_p\,\} \vdash \{\,A_p\,\}\ c_1\ \{\,Q\,\} \qquad \{\,A_p\,\}\ \texttt{call}\ p\ \{\,B_p\,\} \vdash \{\,P\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{coroutine}\ p = c_1\ \texttt{in}\ c_2\ \{\,Q\,\}}$$

(NB : même problème avec la notation $X \vdash Y$ que pour la règle de Clint-Hoare pour le `goto` ; même solution.)

Un exemple de vérification

```
coroutine p =
    begin {Inv ∧ 0 ≤ obs < N}
        while c < N do
            h[obs] := h[obs] + 1;  c := c + 1;
            {Inv} yield p {Inv ∧ 0 ≤ obs < N}
        done
    end
in
    ... obs = 12; {Inv ∧ 0 ≤ obs < N} call p; {Inv} ...
```

L'invariant *Inv* est $\mathtt{c} \leq \mathtt{N} \land \mathtt{c} = \Sigma_{i=0}^{\mathtt{N}-1} \mathtt{h}[i]$.

La précondition A_p du `call` est $Inv \land 0 \leq \mathtt{obs} < \mathtt{N}$.
Elle garantit que l'accès h[obs] est dans les bornes.

La postcondition B_p est *Inv*.

Les *threads* coopératifs

Un modèle simple de *threads* coopératifs :

$$\texttt{run}\ c_1 \parallel \cdots \parallel c_n\ \texttt{end}$$

Les commandes $c_1, \ldots, c_n$ sont exécutées en entrelacement.

Chaque commande fait `yield` pour offrir de passer la main à une autre commande. Entre deux `yield` l'exécution est purement séquentielle.

Le `run` ... `end` termine quand toutes les commandes c_i ont terminé.

Exemple : un modèle producteur-consommateur

```
var full: bool = false; var data: T = null;

run
    while true do                  ||    while true do
        x := produce();            ||        while not full do
        while full do              ||            yield
            yield                  ||        done;
        done;                      ||        y := data;
        data := x;                 ||        full := false;
        full := true;              ||        yield;
        yield                      ||        consume(y)
    done                           ||    done

end
```

Une règle pour les *threads* coopératifs

Une variante symétrisée de la règle de Clint pour les coroutines :

$$\frac{\{\,P\,\}\ \texttt{yield}\ \{\,P\,\} \vdash \{\,P\,\}\ c_i\ \{\,Q\,\} \quad \text{pour } i = 1, \ldots, n}{\{\,P\,\}\ \texttt{run}\ c_1 \parallel \cdots \parallel c_n\ \texttt{end}\ \{\,Q\,\}}$$

La précondition P joue le rôle d'invariant au moment des ≪commutations de contextes≫ d'un `yield` vers le début d'un c_i ou d'un `yield` vers un autre `yield`.

Le calcul peut démarrer avec n'importe lequel des c_i et se terminer par n'importe lequel des c_i.

Une vérification du schéma producteur-consommateur

```
while true do {P}
    x := produce(); {P ∧ R(x)}
    while full do
        yield {P ∧ R(x)}
    done;
    {full = false ∧ P ∧ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true; {P}
    yield {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ∧ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ∧ P}
    yield; {R(y) ∧ P}
    consume(y) {P}
done
```

On se donne un invariant $R(x)$ sur les valeurs de type `T`, tel que
$\{\,\texttt{true}\,\}\; x := \texttt{produce}()\; \{\,R(x)\,\}$ et $\{\,R(x)\,\}\; \texttt{consume}(x)\; \{\,\texttt{true}\,\}$.

Une vérification du schéma producteur-consommateur

```
while true do {P}
    x := produce(); {P ∧ R(x)}
    while full do
        yield {P ∧ R(x)}
    done;
    {full = false ∧ P ∧ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true; {P}
    yield {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ∧ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ∧ P}
    yield; {R(y) ∧ P}
    consume(y) {P}
done
```

On prend comme invariant de la coroutine

$$P \stackrel{def}{=} \texttt{full} = \texttt{true} \Rightarrow R(\texttt{data})$$

Cela montre que les valeurs passées à `consume` satisfont R.

Logiques de séparation pour les opérateurs de contrôle

Un petit langage fonctionnel et impératif

Dans le style des langages ML, en utilisant des références pour représenter l'état mutable.

$e ::= cst \mid x \mid \lambda x.\, e \mid e_1\ e_2$	langage fonctionnel
$\mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	«séquence»
$\mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditionnelle
$\mid \ell$	adresse d'une référence
$\mid \texttt{ref}\ e$	création d'une référence
$\mid\ !\, e \mid e_1 := e_2$	déréférencement, affectation
$\mid \texttt{free}\ e$	désallocation d'une référence

Notation : $e_1; e_2$ est $\texttt{let}\ z = e_1\ \texttt{in}\ e_2$ où z n'est pas libre dans e_2.

Exemple : $\texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in}\ (\texttt{if}\ b\ \texttt{then}\ x := 1\ \texttt{else}\ ()); !\, x$

Triplets de la logique de séparation

$$\{\,P\,\}\; e\; \{\,Q\,\}$$

La précondition P est une assertion.

La postcondition Q est une fonction valeur de $e \mapsto$ assertion.

$$\frac{v \text{ valeur} \qquad P \Rightarrow Q\,v}{\{\,P\,\}\; v\; \{\,Q\,\}} \qquad \frac{\{\,P\,\}\; e_1\; \{\,R\,\} \qquad \forall x,\; \{\,R\,x\,\}\; e_2\; \{\,Q\,\}}{\{\,P\,\}\; \texttt{let}\; x = e_1\; \texttt{in}\; e_2\; \{\,Q\,\}}$$

Assertions de la logique de séparation

Des assertions qui décrivent des morceaux de l'état mémoire (ensemble d'adresses + leurs contenus) :

`emp`	la mémoire est vide
$\langle P \rangle$	la mémoire est vide et la proposition P est vraie
$\ell \mapsto v$	la mémoire se compose d'une case ℓ contenant la valeur v
$\ell \mapsto _$	la mémoire se compose d'une case ℓ (= $\exists v, \ell \mapsto v$)
$P \star Q$	conjonction séparante : la mémoire se décompose en 2 parties disjointes, l'une satisfaisant P, l'autre satisfaisant Q
$P \mathrel{-\!\!\star} Q$	implication séparante (*magic wand*) : si on ajoute à la mémoire une partie satisfaisant P, on obtient une mémoire qui satisfait Q.

Quelques règles de logique de séparation

Les ≪petites≫ règles pour l'état mutable :

$$
\begin{array}{rcl}
\{\,\mathtt{emp}\,\} & \mathtt{ref}\ v & \{\,\lambda\ell.\ \ell \mapsto v\,\} \\
\{\,\ell \mapsto v\,\} & !\,\ell & \{\,\lambda x.\ \langle x = v\rangle \star \ell \mapsto v\,\} \\
\{\,\ell \mapsto _\,\} & \ell := v & \{\,\lambda x.\ \langle x = ()\rangle \star \ell \mapsto v\,\} \\
\{\,\ell \mapsto _\,\} & \mathtt{free}\ \ell & \{\,\lambda x.\ \langle x = ()\rangle\,\}
\end{array}
$$

Se combinent avec la règle d'encadrement (*frame rule*) pour s'appliquer à des états mémoires plus grands :

$$
\frac{\{\,P\,\}\ e\ \{\,Q\,\}}{\{\,P \star R\,\}\ e\ \{\,\lambda x.\ Q\ x \star R\,\}}
$$

Avantages de la logique de séparation

1– On peut raisonner localement sur des codes utilisant des pointeurs sans se soucier des risques d'*aliasing* :

$$\{ \ell_1 \mapsto 1 \star \ell_2 \mapsto v \} \; \ell_1 := 0 \; \{ \ell_1 \mapsto 0 \star \ell_2 \mapsto v \}$$

Pas besoin de traiter le cas $\ell_1 = \ell_2$, la précondition est fausse dans ce cas.

2– La logique garde trace des ressources (mémoire, etc.) et assure qu'on les utilise de manière linéaire ou affine :

✔ $\{ \texttt{emp} \} \; \texttt{let} \; x = \texttt{ref} \; v \; \texttt{in} \; \ldots ; \texttt{free}(x) \; \{ \lambda_.\texttt{emp} \}$

✘ $\{ \texttt{emp} \} \; \texttt{let} \; x = \texttt{ref} \; v \; \texttt{in} \; \ldots ; \texttt{free}(x); !x \; \{ \lambda_.\texttt{emp} \}$ (*use after free*)

✘ $\{ \texttt{emp} \} \; \texttt{let} \; x = \texttt{ref} \; v \; \texttt{in} \; \ldots ; \texttt{free}(x); \texttt{free}(x) \; \{ \lambda_.\texttt{emp} \}$ (*double free*)

✘ $\{ \texttt{emp} \} \; \texttt{let} \; x = \texttt{ref} \; v \; \texttt{in} \; \ldots \; \{ \lambda_.\texttt{emp} \}$ (fuite mémoire)

Retour sur le schéma producteur-consommateur

```
while true do {P}
    x := produce(); {P ✶ R(x)}
    while full do
        yield {P ✶ R(x)}
    done;
    {full = false ✶ P ✶ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true; {P}
    yield {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ✶ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ✶ P}
    yield; {R(y) ✶ P}
    consume(y) {P}
done
```

En logique de séparation, l'invariant R décrit également l'allocation et la libération de ressources :
$\{\,\mathtt{emp}\,\}\ x := \mathtt{produce}()\ \{\,R(x)\,\}$ et $\{\,R(x)\,\}\ \mathtt{consume}(x)\ \{\,\mathtt{emp}\,\}$.

Retour sur le schéma producteur-consommateur

```
while true do {P}
    x := produce(); {P ✶ R(x)}
    while full do
        yield {P ✶ R(x)}
    done;
    {full = false ✶ P ✶ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true; {P}
    yield {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ✶ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ✶ P}
    yield; {R(y) ✶ P}
    consume(y) {P}
done
```

On prend comme invariant $P \stackrel{def}{=} \texttt{if full then } R(\texttt{data}) \texttt{ else emp}$

de sorte que $\texttt{full} = \texttt{false} \star P \iff \texttt{emp}$

et $\texttt{full} = \texttt{true} \star P \iff R(\texttt{data})$.

Retour sur le schéma producteur-consommateur

```
while true do {P}
    x := produce(); {P ✶ R(x)}
    while full do
       yield {P ✶ R(x)}
    done;
    {full = false ✶ P ✶ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true; {P}
    yield {P}
done
```

‖

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ✶ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ✶ P}
    yield; {R(y) ✶ P}
    consume(y) {P}
done
```

On voit le transfert des ressources $R(\texttt{data})$ du producteur vers le consommateur. Cela montre que chaque ressource allouée par `produce` est libérée exactement une fois par `consume`.

Problème avec les opérateurs de contrôle

$$\frac{\{\,P\,\}\ e_1\ \{\,R\,\} \qquad \forall x, \{\,R(x)\,\}\ e_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \{\,Q\,\}}$$

Un opérateur de contrôle comme `call/cc` peut invalider la règle ci-dessus : si e_1 capture sa continuation, e_2 peut être exécutée plusieurs fois, la première fois dans un état mémoire satisfaisant $R(x)$, la deuxième fois pas forcément.

Problème avec les opérateurs de contrôle

(A. Timany, L. Birkedal, *Mechanized relational verification of concurrent programs with continuations*, ICFP 2019.)

$$\begin{aligned} e \stackrel{def}{=} \;& \texttt{let}\; x = \texttt{ref}\; 0\; \texttt{in} \\ & \texttt{let}\; g = f\; ()\; \texttt{in} \\ & x := !\,x + 1; g\; (); !\,x \end{aligned}$$

Si f est une fonction pure renvoyant une fonction pure :

$$\{\, \mathrm{emp} \,\}\, f\; ()\; \{\, \lambda g.\, \{\, \mathrm{emp} \,\}\, g\; ()\; \{\, \lambda_.\mathrm{emp} \,\}\,\}$$

on peut montrer que e s'évalue à 1 :

$$\{\, \mathrm{emp} \,\}\, e\, \{\, \lambda v.\; v = 1 \,\}$$

Problème avec les opérateurs de contrôle

(A. Timany, L. Birkedal, *Mechanized relational verification of concurrent programs with continuations*, ICFP 2019.)

$$e \stackrel{def}{=} \begin{array}{l} \texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in} \\ \texttt{let}\ g = f\ ()\ \texttt{in} \\ x := !\,x + 1; g\ (); !\,x \end{array}$$

Pourtant, si

$$f = \lambda().\texttt{callcc}\,(\lambda k.\ \lambda().\ \texttt{throw}\ k\,(\lambda().()))$$

l'affectation $x := !\,x + 1$ est exécutée deux fois, et $!\,x = 2$ à la fin.

Problème avec les opérateurs de contrôle

(A. Timany, L. Birkedal, *Mechanized relational verification of concurrent programs with continuations*, ICFP 2019.)

$$\begin{aligned} e \overset{def}{=} \; & \texttt{let}\, x = \texttt{ref}\ 0\ \texttt{in} \\ & \texttt{let}\, g = f\ ()\ \texttt{in} \\ & x := !x + 1; g\ (); !x \end{aligned}$$

$$f = \lambda().\texttt{callcc}\,(\lambda k.\,\lambda().\,\texttt{throw}\ k\,(\lambda().()))$$

f est pure au sens où elle ne modifie pas l'état mémoire. Plus précisément, f considérée dans le contexte vide vérifie le contrat $\{\,\mathrm{emp}\,\}\, f\ ()\ \{\,\lambda g.\,\{\,\mathrm{emp}\,\}\ g\ ()\ \{\,\lambda_.\mathrm{emp}\,\}\,\}$.

Une logique pour les programmes complets

L'approche de Timany & Birkedal : définir la logique $\{ P \}\ e\ \{ Q \}$ pour des programmes complets e.

Les règles s'appliquent à des décompositions $e = C[e_1]$,
où C est un contexte d'évaluation et e_1 une expression qui peut se réduire. Elles ressemblent beaucoup aux règles de réduction !

$$\frac{\{ P \}\ C[e[x \leftarrow v]]\ \{ Q \}}{\{ P \}\ C[(\lambda x.\, e)\ v]\ \{ Q \}} \qquad \frac{\{ P \}\ C[e_1]\ \{ Q \}}{\{ P \}\ C[\texttt{if true then}\ e_1\ \texttt{else}\ e_2]\ \{ Q \}}$$

$$\frac{\{ P \}\ C[v\ (\texttt{cont}\ C)]\ \{ Q \}}{\{ P \}\ C[\texttt{callcc}\ v]\ \{ Q \}} \qquad \frac{\{ P \}\ D[v]\ \{ Q \}}{\{ P \}\ C[\texttt{throw}\ (\texttt{cont}\ D)\ v]\ \{ Q \}}$$

Une logique pour les programmes complets

Exemple de vérification :

$$\dfrac{\dfrac{\{\,\mathtt{emp}\,\}\ 5+2\ \{\,\lambda x.\ \langle x = 7\rangle\,\}}{\{\,\mathtt{emp}\,\}\ \mathtt{throw}\ (\mathtt{cont}\ ([\,]+2)\ 5+4)\ \{\,\lambda x.\ \langle x = 7\rangle\,\}}\ (**)}{\{\,\mathtt{emp}\,\}\ \mathtt{callcc}(\lambda k.\ \mathtt{throw}\ k\ 5+4)+2\ \{\,\lambda x.\ \langle x = 7\rangle\,\}}\ (*)$$

(*) utilise la règle du `callcc` avec le contexte $C = [\,] + 2$.

(**) utilise la règle du `throw` avec le contexte $C = [\,]$.

(Voir l'article de Timany et Birkedal pour des exemples plus difficiles !)

Triplets valides dans tous les contextes

Pour faciliter la vérification, on définit les triplets valides dans tous les contextes $\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}$ comme étant ceux qui valident la règle de passage au contexte

$$\frac{\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, R \,\}\!\} \qquad \forall v,\ \{\, R\ v \,\}\ C[v]\ \{\, Q \,\}}{\{\, P \,\}\ C[e]\ \{\, Q \,\}}$$

On peut définir des règles pour $\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}$ qui ressemblent beaucoup aux règles usuelles, pour toutes les expressions *e* sauf `callcc` et `throw`. (Voir l'article.)

Vers une logique pour les gestionnaires d’effets

Les effets définis par le programmeur et leurs gestionnaires devraient permettre un raisonnement plus simple que `call/cc` :

- continuations délimitées,
- que l’on peut spécifier à l’avance par des contrats : précondition sur les arguments / postcondition sur les résultats (un peu comme on spécifie des fonctions);

à condition de

- se restreindre à des continuations à usage unique (*one-shot continuations*).

Problème avec les continuations à usages multiples

$$\frac{\{\, P \,\}\ e_1\ \{\, R \,\} \qquad \forall x, \{\, R(x) \,\}\ e_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \{\, Q \,\}}$$

Cette règle est invalidée si e_1 peut revenir plusieurs fois.

Exemple :

```
handle
    let b = perform Flip in x := !x + 1
with
    val(x) -> x
    Flip(_, k) -> k false; k true
```

`x` est incrémentée deux fois, et non pas une fois comme le prédit la règle `let` avec $P = \mathtt{x} \mapsto 0$ et $Q = \lambda_.\ \mathtt{x} \mapsto 1$.

Protocoles d’effets

(P. E. de Vilhena, F. Pottier, *A separation logic for effect handlers*, POPL 2021.)

Une spécification des comportements des effets. Sert de contrat entre les levées d’effets et les gestionnaires d’effets.

$\Psi ::= \bot$	aucun effet
$\quad \mid\ !\,\vec{x}\,(F\,v)\,\{\,P\,\}.\ ?\,\vec{y}\,(w)\,\{\,Q\,\}$	protocole pour F
$\quad \mid \Psi_1 + \Psi_2$	union de deux protocoles

Le protocole $!\,\vec{x}\,(F\,v)\,\{\,P\,\}.\ ?\,\vec{y}\,(w)\,\{\,Q\,\}$ se lit comme :
≪pour tous les $\vec{x}$, le programme peut lever l’effet F avec la valeur v pourvu que la précondition P soit vraie ; alors il existe des $\vec{y}$ tels que le résultat w de F satisfait la postcondition Q≫.

Exemples de protocoles

L'effet `Abort` (qui ne revient jamais) :

$$! \ (\texttt{Abort}\ ())\ \{\, \texttt{true} \,\}.\ ?\, y\ (y)\ \{\, \texttt{false} \,\}$$

L'effet `Next` simulant un compteur :

$$!\, n\ (\texttt{Next}\ ())\ \{\, \mathit{Count}\ n \,\}.\ ?\ \ (n)\ \{\, \mathit{Count}\ (n+1) \,\}$$

Le prédicat abstrait *Count n* garde trace de la valeur courante du compteur.

Les effets `Get` et `Set` simulant une référence :

$$\begin{array}{ll} & !\, v\ (\texttt{Get}\ ())\ \{\, \mathit{State}\ v \,\}.\ ?\ \ (v)\ \{\, \mathit{State}\ v \,\} \\ + & !\, v\, v'\ (\texttt{Set}\ v')\ \{\, \mathit{State}\ v \,\}.\ ?\ \ (())\ \{\, \mathit{State}\ v' \,\} \end{array}$$

Le prédicat abstrait *State v* garde trace de la valeur courante de la pseudo-référence.

Un quadruplet de Hoare

$$\{\, P \,\}\; e \;\langle\, \Psi \,\rangle\; \{\, Q \,\}$$

Le protocole Ψ joue le rôle d'une postcondition supplémentaire.

En particulier, $\{\, P \,\}\; e \;\langle\, \bot \,\rangle\; \{\, Q \,\}$ garantit que e ne lève pas d'effets non gérés.

Le protocole «distribue» sur les calculs qui ne gèrent pas d'effets :

$$\frac{v \text{ valeur} \qquad P \Rightarrow Q\, v}{\{\, P \,\}\; v \;\langle\, \Psi \,\rangle\; \{\, Q \,\}}$$

$$\frac{\{\, P \,\}\; e_1 \;\langle\, \Psi \,\rangle\; \{\, R \,\} \qquad \forall x,\ \{\, R\, x \,\}\; e_2 \;\langle\, \Psi \,\rangle\; \{\, Q \,\}}{\{\, P \,\}\; \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \;\langle\, \Psi \,\rangle\; \{\, Q \,\}}$$

Levée d'effets

$$\frac{P \Rightarrow \Psi \ \texttt{allows} \ (F\,v')\,\{\,Q\,\}}{\{\,P\,\}\,\texttt{perform}\,F\,v'\,\langle\,\Psi\,\rangle\,\{\,Q\,\}}$$

$\bot \ \texttt{allows} \ (F\,v')\,\{\,Q\,\}$ est toujours faux.

$\Psi_1 + \Psi_2 \ \texttt{allows} \ (F\,v')\,\{\,Q\,\}$ est la disjonction
$\Psi_1 \ \texttt{allows} \ (F\,v')\,\{Q\} \vee \Psi_2 \ \texttt{allows} \ (F\,v')\,\{\,Q\,\}$.

$!\,\vec{x}\,(F\,v)\,\{\,A\,\}.\,?\,\vec{y}\,(w)\,\{\,B\,\} \ \texttt{allows} \ (F\,v')\,\{\,Q\,\}$ est vrai si

$$\exists\vec{x},\ \langle v' = v\rangle \ast A \ast (\forall\vec{y},\ B \mathrel{-\!\!\ast} Q(w))$$

c.à.d. : en choisissant $\vec{x}$ pour égaliser l'argument effectif v' et le paramètre formel v, la précondition A de F doit être satisfaite, et pour tout choix de $\vec{y}$ qui satisfait la postcondition B de F, la postcondition $Q(w)$ est vraie.

Gestion d’effets

$$\frac{\{\,P\,\}\;e\;\langle\,\Psi\,\rangle\;\{\,Q\,\} \qquad \mathit{isHandler}\;\langle\,\Psi\,\rangle\;\{\,Q\,\}\;(e_{val}, e_{eff})\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}{\{\,P\,\}\;\texttt{handle}\;e\;\texttt{with}\;e_{val}, e_{eff}\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}$$

Comme toujours, le but du gestionnaire est de transformer les résultats $\langle\Psi\rangle\;\{\,Q\,\}$ du calcul géré e en résultats $\langle\Psi'\rangle\;\{\,Q'\,\}$.

Si e termine normalement, sa valeur v satisfait Q, et $e_{val}\;v$ est exécuté. Ce calcul doit donc vérifier

$$\{\,Q(v)\,\}\;e_{val}\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}$$

Gestion d’effets

$$\frac{\{\,P\,\}\; e \;\langle\,\Psi\,\rangle\;\{\,Q\,\} \qquad \mathit{isHandler}\;\langle\,\Psi\,\rangle\;\{\,Q\,\}\;(e_{val}, e_{eff})\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}{\{\,P\,\}\;\texttt{handle}\; e \;\texttt{with}\; e_{val}, e_{eff}\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}$$

Si e termine en levant une valeur d’effet v avec continuation k, $e_{eff}\ v\ k$ est exécuté, et doit satisfaire

$$\{\,R\,\}\; e_{eff}\ v\ k\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}$$

La précondition R pourrait dire quelque chose comme : si v est $F\ v'$ et le protocole Ψ associe à F la précondition A et la postcondition B, alors

- v' satisfait A ;
- k est une fonction avec pré B et post $\langle\,\Psi'\,\rangle\;\{\,Q'\,\}$.

Gestion d'effets

$$\frac{\{\, P \,\}\; e \;\langle\, \Psi \,\rangle\; \{\, Q \,\} \qquad \mathit{isHandler}\; \langle\, \Psi \,\rangle\; \{\, Q \,\}\; (e_{val}, e_{eff})\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\}}{\{\, P \,\}\; \texttt{handle}\; e\; \texttt{with}\; e_{val}, e_{eff}\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\}}$$

Plus simplement, R dit que v et k sont des valeurs d'effet et des continuations que le protocole Ψ autorise :

$$R \stackrel{def}{=} \Psi \;\texttt{allows}\; v\, \{\, \lambda w.\, \{\, \texttt{emp} \,\}\; k\, w\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\} \,\}$$

Avantage supplémentaire : la formalisation Iris de cette théorie utilise des ≪triplets non persistants≫, et donc $\{\, \texttt{emp} \,\}\; k\, w\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\}$ donne la permission d'appliquer la continuation k une seule fois !

Gestion d'effets

$$\frac{\{\,P\,\}\; e\; \langle\,\Psi\,\rangle\; \{\,Q\,\} \qquad \mathit{isHandler}\; \langle\,\Psi\,\rangle\; \{\,Q\,\}\; (e_{val}, e_{eff})\; \langle\,\Psi'\,\rangle\; \{\,Q'\,\}}{\{\,P\,\}\; \texttt{handle}\; e\; \texttt{with}\; e_{val}, e_{eff}\; \langle\,\Psi'\,\rangle\; \{\,Q'\,\}}$$

En résumé, on a

$$\begin{array}{l}
\mathit{isHandler}\; \langle\,\Psi\,\rangle\; \{\,Q\,\}\; (e_{val}, e_{eff})\; \langle\,\Psi'\,\rangle\; \{\,Q'\,\} \;\stackrel{def}{=} \\
\qquad (\forall v,\; \{\,Q(v)\,\}\; e_{val}\; v\; \langle\,\Psi'\,\rangle\; \{\,Q'\,\}\,) \\
\quad \wedge \;\; (\forall v, k,\; \{\,\Psi\;\; \texttt{allows}\;\; v\; \{\,\lambda w.\, \{\,\texttt{emp}\,\}\; k\; w\; \langle\,\Psi'\,\rangle\; \{\,Q'\,\}\,\}\,\} \\
\qquad\qquad e_{eff}\; v\; k \\
\qquad\qquad \langle\,\Psi'\,\rangle\; \{\,Q'\,\}
\end{array}$$

Ceci décrit un gestionnaire superficiel (*shallow handler*).
Pour un gestionnaire profond, voir l'article.

Point d’étape

Point d'étape

Conçues initialement pour le contrôle structuré, les logiques de programmes (logique de Hoare, logique de séparation) s'étendent assez facilement

- au `goto`, aux sorties `break`, `return`, etc., et aux exceptions ;
- aux coroutines et aux *threads* coopératifs ;
- aux fonctions du premier ordre. (pas traité aujourd'hui)

D'autre traits linguistiques sont plus problématiques :

- les fonctions d'ordre supérieur ; (pas traité aujourd'hui)
- les opérateurs de contrôle.

On est aidé par la structure supplémentaire qu'offrent les gestionnaires d'effets par rapport à `call/cc`.

Bibliographie

Bibliographie

Introduction à la logique de Hoare et à la logique de séparation :

- Mon cours 2020–2021 ≪Logiques de programmes≫, séances 1 à 3.

Une logique pour `call/cc` :

- Amin Timany, Lars Birkedal : *Mechanized Relational Verification of Concurrent Programs with Continuations*, PACMPL 3(ICFP), 2019.

Une logique pour les effets et les gestionnaires d'effets :

- Paulo Emílio de Vilhena, François Pottier : *A Separation Logic for Effect Handlers*, PACMPL 5(POPL), 2021.

FIN