Logiques de programmes, septième cours

Logiques pour les langages fonctionnels et l’ordre supérieur

Xavier Leroy

2021-04-15

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

Quelles logiques de programmes pour les langages fonctionnels ?

Faut-il une logique de programmes pour un langage fonctionnel ?

Non, si les fonctions définissables dans le langage sont des fonctions de la logique ambiante :

- fonctions totales (pas de divergence, pas d'erreurs)
- sans effets (pas de traits impératifs).

Exemple : en Coq ou en Agda, les fonctions que l'on peut écrire sont des objets de la logique ambiante (théorie des types).

Dans ce cas, des énoncés et des démonstrations de la logique ambiante «fonctionnent» aussi bien que des triplets de Hoare :

$$\forall x, P\,x \Rightarrow Q\,x\,(f\,x) \text{ au lieu de } \{\,P\,\}\ f\ x\ \{\,Q\,\}$$

Faut-il une logique de programmes pour un langage fonctionnel ?

Oui, probablement, si le langage fonctionnel est doté d'effets :

- divergence ;
- erreurs à l'exécution ;
- état mutable, entrées/sorties ;
- exceptions, continuations, effets algébriques.

On peut raisonner «à la main» sur des programmes fonctionnels avec effets, p.ex. via une traduction monadique.

Mais une logique de programmes adaptée donnera des principes de spécification et de raisonnement de plus haut niveau.

Exemple : raisonner sur l'état mutable

On peut représenter le calcul impératif en Coq sous forme de transformateurs d'état : des fonctions pures

$$\text{état} \ll\text{avant}\gg \;\rightarrow\; \text{valeur} \times \text{état} \ll\text{après}\gg$$

Énoncer et démontrer des propriétés de ces calculs est pénible :

```
forall x s, valid x s ->
let (y, s') := f x s in
~valid y s /\ valid y s' /\ s' x = 0 /\ s' y = s x /\
(forall l, l <> x -> l <> y -> s' l = s l).
```

En logique de séparation, on écrirait simplement

$$\forall x, \{\, x \mapsto n \,\}\; f\; x\; \{\, \lambda y.\; x \mapsto 0 \star y \mapsto n \,\}$$

Exemple : raisonner sur la non-terminaison

Plusieurs représentations possibles pour les calculs qui peuvent ne pas terminer (récursion générale) (cours du 2020-01-30).

Par exemple : la monade de partialité de Capretta (2005)

```
CoInductive delay (A: Type) : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

Le triplet faible $\{ P \}\ c\ \{ Q \}$ devient $P \to$ `safe` $c\ Q$, où `safe` est le prédicat coinductif suivant :

```
CoInductive safe {A: Type}: delay A -> (A -> Prop) -> Prop :=
  | safe_now: forall a Q, Q a -> safe (now a) Q
  | safe_later: forall c Q, safe c Q -> safe (later c) Q
```

Plan du cours

Deux angles d'attaque :

- Comment étendre la logique de Hoare et la logique de séparation pour traiter les fonctions, y compris l'ordre supérieur et les fonctions comme valeurs de première classe ?
  Exemple : Iris.
- Comment utiliser l'ordre supérieur et les types dépendants pour présenter autrement les logiques de programmes ?
  Exemples : F*, CFML.

Procédures et fonctions du premier ordre en logique de Hoare et en logique de séparation

Les procédures en logique de Hoare

Une des premières extensions de la logique de Hoare.

Une motivation pratique : vérifier Quicksort. (Foley et Hoare, 1971)

Un principe de modularité du raisonnement :

> Les procédures permettent de réutiliser du code dans plusieurs contextes d'appel. Peut-on réutiliser la vérification de ce code ? (au lieu de le revérifier dans chaque contexte d'appel)

Des questions de sémantique (des liaisons de variables, des mécanismes de passage de paramètres, etc).

Une présentation antichronologique

Les règles pour les procédures en logique de Hoare sont compliquées car il faut contrôler les mutations sur les variables.

Suivant Parkinson, Bornat et Calcagno (2006) :

- Nous commençons par ajouter des procédures et des fonctions au langage PTR, où les variables sont immuables mais peuvent être des références vers des cases mémoire mutables, et utilisons la logique de séparation pour spécifier tout cela.
- Ensuite et pour mémoire, nous étendons IMP avec des procédures et esquissons les règles correspondantes en logique de Hoare.

Les fonctions dans le langage PTR

Commandes : $c ::= \ldots$

$$\mid \texttt{let}\ f\ (\vec{x}) = c\ \texttt{in}\ c' \quad \text{définition de fonction}$$

$$\mid f\ (\vec{a}) \quad \text{appel de fonction}$$

Il s'agit de fonctions impératives, à la manière du langage C : elles peuvent modifier l'état tout en renvoyant une valeur.

Exemple : la fonction *minmaxplus*.

$$\begin{array}{l} \texttt{let}\ \mathit{minmaxplus}\ (x, y, m, M) = \\ \quad \texttt{if}\ x < y\ \texttt{then}\ (\texttt{set}(m, x); \texttt{set}(M, y)) \\ \qquad\qquad \texttt{else}\ (\texttt{set}(m, y); \texttt{set}(M, x)); \\ \quad x + y \end{array}$$

Spécifier une fonction

Spécification de la forme $\{ P \} f (\vec{x}) \{ Q \}$ où P et Q sont des formules de logique de séparation.

Exemple : la fonction *minmaxplus*.

$$\{ m \mapsto _ \star M \mapsto _ \}$$
$$\mathit{minmaxplus}\,(x, y, m, M)$$
$$\{ \lambda v.\langle v = x + y \rangle \star m \mapsto \min(x, y) \star M \mapsto \max(x, y) \}$$

Exemple : une fonction $\mathit{incr}(d)$ qui incrémente un compteur global c de la valeur d et renvoie l'ancienne valeur de c.

$$\forall \alpha,\ \{ c \mapsto \alpha \}\ \mathit{incr}\,(d)\ \{ \lambda v.\ \langle v = \alpha \rangle \star c \mapsto \alpha + d \}$$

Les règles pour les fonctions

Un contexte Γ = un ensemble de spécifications de fonctions.

Appel de fonction :

$$\frac{(\{\, P \,\}\, f\ (\vec{x})\ \{\, Q \,\}) \in \Gamma}{\Gamma \vdash \{\, P[\vec{x} \leftarrow [\![\vec{a}]\!]] \,\}\, f\ (\vec{a})\ \{\, Q[\vec{x} \leftarrow [\![\vec{a}]\!]] \,\}}$$

Définition de fonction :

$$\frac{\begin{array}{c}\Gamma' = \Gamma,\ \{\, P \,\}\, f\ (\vec{x})\ \{\, Q \,\} \\ \forall \vec{x},\ \Gamma' \vdash \{\, P \,\}\ c\ \{\, Q \,\} \\ \Gamma' \vdash \{\, P' \,\}\ c'\ \{\, Q' \,\}\end{array}}{\Gamma \vdash \{\, P' \,\}\ \texttt{let}\ \ f\ (\vec{x}) = c\ \ \texttt{in}\ \ c'\ \{\, Q' \,\}}$$

La règle de Hoare pour la récursion

$$\frac{\Gamma, \{\, P \,\}\, f\,()\, \{\, Q \,\} \vdash \{\, P \,\}\, c\, \{\, Q \,\}}{\Gamma \vdash \{\, P \,\}\, f\,()\, \{\, Q \,\}}$$

Lecture ≪coinductive≫ : on peut utiliser la conclusion comme hypothèse pourvu qu'elle soit gardée par une ou plusieurs règles.

Lecture ≪comptage de pas≫ : pour montrer que le triplet $\{\, P \,\}\, f\,()\, \{\, Q \,\}$ est valide pour n étapes de calcul, on peut supposer qu'il l'est pour $j < n$ étapes de calcul.

Lecture ≪modale≫ : c'est la règle de Löb pour la modalité $\triangleright$

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

Un exemple de vérification

Avec la spécification $\{ x \mapsto _ \}\ \mathit{slowset}\ (x, n)\ \{ x \mapsto n \}$

```
let slowset (x, n) =                          { x ↦ _ }
    if n = 0 then
        set(x, 0)                             { x ↦ 0 }
    else
        slowset (x, n − 1);                   { x ↦ n − 1 }
        let v = get(x) in set(v, x + 1)       { x ↦ n }
in
                                              { a ↦ _ ∗ b ↦ _ }
    slowset(a, 2);                            { a ↦ 2 ∗ b ↦ _ }
    slowset(b, 3)                             { a ↦ 2 ∗ b ↦ 3 }
```

Commandes :

$$
\begin{aligned}
c ::= {} & \ldots \\
& \mid \texttt{local}\ x\ \texttt{in}\ c && \text{variable locale} \\
& \mid \texttt{let}\ f\ (\texttt{var}\ \vec{x}; \texttt{val}\ \vec{y}) = c\ \texttt{in}\ c' && \text{définition de procédure} \\
& \mid f\ (\vec{x}, \vec{a}) && \text{appel de procédure}
\end{aligned}
$$

Les paramètres $\vec{x}$ sont passés par référence.
Les arguments correspondants sont des variables.

Les paramètres $\vec{y}$ sont passés par valeur.
Les arguments correspondants sont des expressions.

Exemple : minimum et maximum.

$$
\begin{array}{l}
\texttt{let}\ \ \mathit{minmax}\,(\texttt{var}\ \ m, M; \texttt{val}\ \ x, y) = \\
\quad \texttt{if}\ \ x < y\ \ \texttt{then}\ \ (m := x; M := y)\ \ \texttt{else}\ \ (m := y; M := x)
\end{array}
$$

Règles pour les procédures avec paramètres

La spécification d'une procédure est un triplet enrichi par des informations sur les variables utilisées :

$$\{\, P \,\}\, f(\mathtt{var}\ \vec{x}, \mathtt{val}\ \vec{y})\ [\mathtt{uses}\ \bar{u}, \mathtt{modifies}\quad \bar{v}]\ \{\, Q \,\}$$

$\bar{u}$ est l'ensemble des variables non locales libres dans f

$\bar{v}$ est l'ensemble des variables non locales modifiées par f.

Appel de procédure :

$$\frac{(\{\, P \,\}\, f(\mathtt{var}\ \vec{x}, \mathtt{val}\ \vec{y})\ [\mathtt{uses}\ \bar{u}, \mathtt{modifies}\quad \bar{v}]\ \{\, Q \,\}) \in \Gamma \qquad \vec{w} \cap (\bar{u} \cup \bar{v}) = \emptyset}{\Gamma \vdash \{\, \vec{\alpha} = \vec{a} \wedge P[\vec{x} \leftarrow \vec{w}, \vec{y} \leftarrow \vec{\alpha}] \,\}\ f\ (\vec{w}, \vec{a})\ \{\, Q[\vec{x} \leftarrow \vec{w}, \vec{y} \leftarrow \vec{\alpha}] \,\}}$$

Règles pour les procédures avec paramètres

Définition de procédure :

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \{\, P \,\}\, f\ (\texttt{var}\ \ \vec{x}, \texttt{val}\ \ \vec{y})\ [\texttt{uses}\ \ \bar{u}, \texttt{modifies}\ \ \bar{v}]\ \{\, Q \,\} \\ \bar{u} = \mathit{free}_\Gamma(c) \setminus (\vec{x} \cup \vec{y}) \qquad \bar{v} = \mathit{mods}_\Gamma(c) \setminus (\vec{x} \cup \vec{y}) \\ \vec{z} \cap \mathit{free}(P, Q, c, \vec{x}, \vec{y}) = \emptyset \\ \Gamma' \vdash \{\, P \,\}\ \texttt{local}\ \ \vec{z}\ \ \texttt{in}\ \ \vec{z} := \vec{y}; c[\vec{y} \leftarrow \vec{z}]\ \{\, Q \,\} \\ \Gamma' \vdash \{\, P' \,\}\ c'\ \{\, Q' \,\}\end{array}}{\Gamma \vdash \{\, P' \,\}\ \texttt{let}\ \ f\ (\texttt{var}\ \ \vec{x}; \texttt{val}\ \ \vec{y}) = c\ \ \texttt{in}\ \ c'\ \{\, Q' \,\}}$$

Règles pour les variables locales

La règle correcte (sémantique avec portée statique) :

$$\frac{\{\, P \,\}\ c[x \leftarrow y]\ \{\, Q \,\} \qquad y \notin \mathit{free}(c, P, Q)}{\{\, P \,\}\ \texttt{local}\ x\ \texttt{in}\ c\ \{\, Q \,\}}$$

Une règle tentante mais fausse (portée dynamique) :

$$\frac{\{\, P[x \leftarrow y] \,\}\ c\ \{\, Q[x \leftarrow y] \,\} \qquad y \notin \mathit{free}(c, P, Q)}{\{\, P \,\}\ \texttt{local}\ x\ \texttt{in}\ c\ \{\, Q \,\}}$$

Fonctions comme valeurs de première classe en logique de séparation

PTR avec des fonctions de première classe

$$
\begin{array}{llll}
\text{Expressions :} & a ::= \ldots & & \\
& \quad \mid \texttt{rec}\ f\ x = c & & \text{abstraction de fonction} \\
\text{Commandes :} & c ::= a \mid \ldots & & \\
& \quad \mid a_1\ a_2 & & \text{application de fonction}
\end{array}
$$

Une fonction non récursive $\lambda x.\, c$ est traitée comme une fonction récursive $\texttt{rec}\ f\ x = c$ avec f non libre dans c.

Sémantique : la β-réduction habituelle.

$$(\texttt{rec}\ f\ x = c)\ a/h \ \rightarrow\ c[x \leftarrow [\![a]\!], f \leftarrow \texttt{rec}\ f\ x = c]/h$$

Les triplets de Hoare comme assertions

Assertions, préconditions :

$$P ::= \langle A \rangle \mid \mathtt{emp} \mid l \mapsto v \mid P_1 \star P_2 \mid \dots$$
$$\mid \{ P \}\, c\, \{ Q \} \qquad \text{triplet de Hoare}$$

Postconditions :

$$Q ::= \lambda v.\, P$$

Les assertions ≪triplets≫ sont duplicables :

$$\{ P \}\, c\, \{ Q \} = \{ P \}\, c\, \{ Q \} \star \{ P \}\, c\, \{ Q \}$$

Règles pour les fonctions

Abstraction récursive :

$$\frac{\begin{array}{l}\forall v, \{\, P \,\}\ (\texttt{rec}\ f\ x = c)\ v\ \{\, Q \,\} \Rightarrow \\ \qquad\qquad \forall v, \{\, P \,\}\ c[x \leftarrow v, f \leftarrow \texttt{rec}\ f\ x = c]\ \{\, Q \,\}\end{array}}{\forall v, \{\, P \,\}\ (\texttt{rec}\ f\ x = c)\ v\ \{\, Q \,\}}$$

Abstraction non récursive (règle dérivée) :

$$\frac{\{\, P \,\}\ c[x \leftarrow v]\ \{\, Q \,\}}{\{\, P \,\}\ (\lambda x.c)\ v\ \{\, Q \,\}}$$

Déplacement du triplet de/vers la précondition :

$$\frac{(\forall \vec{v},\ \{\, P_1 \,\}\ c_1\ \{\, Q_1 \,\}) \Rightarrow \{\, P_2 \,\}\ c_2\ \{\, Q_2 \,\}}{\overline{\{\, (\forall \vec{v},\ \{\, P_1 \,\}\ c_1\ \{\, Q_1 \,\}) \star P_2 \,\}\ c_2\ \{\, Q_2 \,\}}}$$

Spécifier une fonction d'ordre supérieur

Prenons la fonction $app = \lambda f. f\ 0$.

On voudrait lui donner la spécification suivante :
≪si f est à valeurs positives, $app\ f$ renvoie un nombre positif≫.

En notant $Q = \lambda x.\ \langle x \geq 0 \rangle$ la postcondition ≪renvoie un nombre positif≫, on a bien

$$\frac{(\forall v, \{\,\mathtt{emp}\,\}\ f\ v\ \{\,Q\,\}) \Rightarrow \{\,\mathtt{emp}\,\}\ f\ 0\ \{\,Q\,\}}{\{\,\forall v, \{\,\mathtt{emp}\,\}\ f\ v\ \{\,Q\,\}\,\}\ app\ f\ \{\,Q\,\}}$$

Représenter un objet avec état local

```
class Counter {
    private int val;
    Counter() { val = 0 }
    int curr() { return val; }
    void incr() { val += 1; }
}
```

Une implémentation dans notre langage PTR :

$$
\begin{array}{l}
\texttt{let}\ \ \mathit{mkpair} = \lambda x.\ \lambda y. \\
\quad \texttt{let}\ \ p = \texttt{alloc}(2)\ \ \texttt{in}\ \ \texttt{set}(p, x); \texttt{set}(p+1, y); p\ \ \texttt{in} \\
\texttt{let}\ \ \mathit{counter} = \lambda_. \\
\quad \texttt{let}\ \ \mathit{val} = \texttt{alloc}(1)\ \ \texttt{in} \\
\quad \mathit{mkpair}\ (\lambda_.\ \texttt{get}(\mathit{val})) \\
\qquad\qquad (\lambda_.\ \texttt{let}\ \ n = \texttt{get}(\mathit{val})\ \ \texttt{in}\ \ \texttt{set}(\mathit{val}, n+1))
\end{array}
$$

Prédicat de représentation

On définit le prédicat $\mathit{Counter}(p, n)$, «à l'adresse p il y a un compteur dont la valeur courante est n» comme

$$\begin{array}{l} \exists \mathit{curr}, \mathit{incr}, \mathit{val},\ p \mapsto \mathit{curr} \ast p + 1 \mapsto \mathit{incr} \ast \mathit{val} \mapsto n \\ \qquad\qquad \ast\ \{\, \mathit{val} \mapsto n \,\}\ \mathit{curr}\ ()\ \{\, \lambda v.\ \langle v = n \rangle \ast \mathit{val} \mapsto n \,\} \\ \qquad\qquad \ast\ \{\, \mathit{val} \mapsto n \,\}\ \mathit{incr}\ ()\ \{\, \lambda_.\ \mathit{val} \mapsto n + 1 \,\} \end{array}$$

On peut alors montrer

$$\begin{array}{rcl} \{\, \mathtt{emp} \,\} & \mathit{counter}\ () & \{\, \lambda p.\ \mathit{Counter}(p, 0) \,\} \\ \{\, \mathit{Counter}(p, n) \,\} & \mathtt{get}(p)\ () & \{\, \lambda v.\ \langle v = n \rangle \ast \mathit{Counter}(p, n) \,\} \\ \{\, \mathit{Counter}(p, n) \,\} & \mathtt{get}(p + 1)\ () & \{\, \lambda_.\ \mathit{Counter}(p, n + 1) \,\} \end{array}$$

Correction sémantique de la règle pour la récursion

$$\frac{\forall v, \{\,P\,\}\ (\texttt{rec}\ \ f\ x = c)\ v\ \{\,Q\,\} \Rightarrow \quad \forall v, \{\,P\,\}\ c[x \leftarrow v, f \leftarrow \texttt{rec}\ \ f\ x = c]\ \{\,Q\,\}}{\forall v, \{\,P\,\}\ (\texttt{rec}\ \ f\ x = c)\ v\ \{\,Q\,\}}$$

Suivant notre approche sémantique usuelle, pour montrer la conclusion, on s'intéresse aux réductions de la commande :

$$(\texttt{rec}\ \ f\ x = c)\ v/h \rightarrow c[x \leftarrow v, f \leftarrow \texttt{rec}\ \ f\ x = c]$$

La prémisse nous donne un triplet sémantique pour $c[x \leftarrow v, f \leftarrow \texttt{rec}\ \ f\ x = c]$, mais seulement si nous avons déjà montré

$$\forall v, \{\,P\,\}\ (\texttt{rec}\ \ f\ x = c)\ v\ \{\,Q\,\}$$

c.à.d. le résultat désiré ! La démonstration tourne en rond !

Le comptage de pas à la rescousse

Idée : dans la définition du triplet de Hoare sémantique

$$\{\!\{\, P \,\}\!\}\ c\ \{\!\{\, Q \,\}\!\} = \forall n,\ h,\ P\ h \Rightarrow \mathtt{Safe}^n\ c\ h\ Q$$

Un appel de fonction dans c prend une étape. Donc la fonction appelée a besoin d'être sûre pour $n - 1$ étapes au plus.

Donc, les triplets de Hoare apparaissant dans la précondition P ont juste besoin d'être vrais ≪à profondeur $n - 1$≫.

Le comptage de pas à la rescousse

Une réalisation de cette idée : on indexe les assertions par un nombre de pas n. Pour les assertions usuelles ce nombre de pas est ignoré :

$$\langle A \rangle\ h\ n = Dom(h) = \emptyset \wedge A$$

$$(\ell \mapsto v)\ h\ n = Dom(h) = \{\ell\} \wedge h\ \ell = v$$

mais il est pris en compte pour les assertions ≪triplet≫

$$(\{\,P\,\}\ c\ \{\,Q\,\})\ h\ 0 = Dom(h) = \emptyset$$

$$(\{\,P\,\}\ c\ \{\,Q\,\})\ h\ (n+1) = Dom(h) = \emptyset \wedge \forall h',\ P\ h'\ n \Rightarrow \mathtt{Safe}^{n+1}\ c\ h'\ Q$$

Le triplet sémantique est alors

$$\{\!\{\,P\,\}\!\}\ c\ \{\!\{\,Q\,\}\!\} = \forall n > 0, \forall h,\ P\ h\ (n-1) \Rightarrow \mathtt{Safe}^{n}\ c\ h\ Q$$

La modalité $\triangleright$ à la rescousse

Une alternative au comptage explicite de pas est d'utiliser une logique modale avec la modalité $\triangleright$ («plus tard»).

Cette modalité permet de faire des démonstrations par récurrence de Löb :

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

mais aussi de définir des prédicats récursifs de la forme

$$P\,x = \ldots \triangleright P\,y \ldots \triangleright P\,z \ldots$$

La modalité $\triangleright$ à la rescousse

En particulier, on peut définir le prédicat $\texttt{Safe}\ c\ h\ Q$
(«si c/h termine, c'est sur un état satisfaisant Q»)
sans comptage de pas, juste comme

$$\begin{aligned}\texttt{Safe}\ c\ h\ Q = {} & (c = a \Rightarrow Q\ [\![a]\!]\ h) \\ & \wedge (c/h \not\to \texttt{err}) \\ & \wedge (\forall c', h', c/h \to c'/h' \Rightarrow \triangleright\texttt{Safe}\ c'\ h'\ Q)\end{aligned}$$

Cette définition de $\texttt{Safe}$ et du triplet sémantique valide la règle pour les fonctions récursives $\texttt{rec}\ \ f\ x = c$, par récurrence de Löb.

Des règles plus puissantes

Pour les règles correspondant à une «vraie» étape de calcul, on peut affaiblir la précondition de P à $\triangleright P$.
(Permet de démontrer plus de résultats par récurrence de Löb.)

$$\frac{\forall v, \{\, P \,\}\, (\texttt{rec}\ f\ x = c)\ v\ \{\, Q \,\} \Rightarrow \quad \forall v, \{\, P \,\}\ c[x \leftarrow v, f \leftarrow \texttt{rec}\ f\ x = c]\ \{\, Q \,\}}{\forall v, \{\, \triangleright P \,\}\, (\texttt{rec}\ f\ x = c)\ v\ \{\, Q \,\}}$$

$$\frac{\{\, P \,\}\ c[x \leftarrow v]\ \{\, Q \,\}}{\{\, \triangleright P \,\}\ (\lambda x.c)\ v\ \{\, Q \,\}}$$

$$\{\, \triangleright \ell \mapsto v \,\}\ \texttt{get}(\ell)\ \{\, \lambda v'.\ \langle v' = v \rangle \ast \ell \mapsto v \,\}$$

$$\{\, \triangleright \ell \mapsto _ \,\}\ \texttt{set}(\ell, v)\ \{\, \lambda _.\ \ell \mapsto v \,\}$$

CFML : raisonner sur des programmes ML avec des formules caractéristiques

Formules caractéristiques pour programmes purs

La formule caractéristique $[\![t]\!]$ d'un terme t est son calcul de plus faible précondition : $[\![t]\!]\ Q = wp(t, Q)$.

$$[\![t]\!] : \underbrace{(\lceil \tau \rceil \to \texttt{Prop})}_{\text{postcondition}} \to \underbrace{\texttt{Prop}}_{\text{précondition}} \qquad \text{si } t : \tau$$

Quelques cas :

$$\begin{aligned}
[\![v]\!] &= \lambda Q.\ Q\ \lceil v \rceil \\
[\![\texttt{fail}]\!] &= \lambda Q.\ \bot \\
[\![\texttt{let}\ x = t\ \texttt{in}\ t']\!] &= \lambda Q.\ \exists R.\ [\![t]\!]\ R \wedge (\forall x, R\ x \Rightarrow [\![t']\!]\ Q) \\
[\![\texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2]\!] &= \lambda Q.\ (\lceil v \rceil \Rightarrow [\![t_1]\!]\ Q) \wedge (\neg \lceil v \rceil \Rightarrow [\![t_2]\!]\ Q)
\end{aligned}$$

Formules caractéristiques pour programmes purs

La vraie définition utilise des combinateurs pour refléter la structure du programme dans la formule caractéristique :

$$[\![v]\!] = \mathtt{Ret}\ \lceil v \rceil \qquad [\![f\ v]\!] = \mathtt{App}\ \lceil f \rceil\ \lceil v \rceil \qquad [\![\mathtt{fail}]\!] = \mathtt{Fail}$$

$$[\![\mathtt{let}\ \ x = t\ \ \mathtt{in}\ \ t']\!] = \mathtt{Let}\ x = [\![t]\!]\ \mathtt{In}\ [\![t']\!]$$

$$[\![\mathtt{if}\ \ v\ \ \mathtt{then}\ \ t_1\ \ \mathtt{else}\ \ t_2]\!] = \mathtt{If}\ \lceil v \rceil\ \mathtt{Then}\ [\![t_1]\!]\ \mathtt{Else}\ [\![t_2]\!]$$

où les combinateurs sont définis par :

$$\mathtt{Ret}\ V = \lambda Q.\ Q\ V \qquad \mathtt{App}\ F\ V = \mathit{AppReturns}\ F\ V \qquad \mathtt{Fail} = \lambda Q.\ \bot$$

$$\mathtt{Let}\ x = F\ \mathtt{In}\ F' = \lambda Q.\ \exists R,\ F\ R \wedge (\forall x,\ R\ x \Rightarrow F'\ Q)$$

$$\mathtt{If}\ V\ \mathtt{Then}\ F\ \mathtt{Else}\ F' = \lambda Q.\ (V \Rightarrow F\ Q) \wedge (\neg V \Rightarrow F'\ Q)$$

Exemple de formule caractéristique

```
let rec half x =
  if x = 0 then 0 else if x = 1 then fail
  else let y = half (x - 2) in y + 1
```

Le corps de la fonction `half` devient

$$\texttt{If}\ x = 0\ \texttt{Then}\ \texttt{Ret}\ 0\ \texttt{Else}\ \texttt{If}\ x = 1\ \texttt{Then}\ \texttt{Fail}$$
$$\texttt{Else}\ \texttt{Let}\ y = \texttt{App}\ \mathit{half}\ (x - 2)\ \texttt{In}\ \texttt{Ret}\ (y + 1)$$

c'est-à-dire

$$\begin{aligned}&\lambda Q.\ (x = 0 \Rightarrow Q\ 0) \wedge (x \neq 0 \Rightarrow \\ &\quad (((x = 1) \Rightarrow \bot) \wedge (x \neq 1 \Rightarrow \\ &\qquad \exists R,\ \mathit{AppReturns}\ \mathit{half}\ (x - 2)\ R \wedge (\forall y, R\ y \Rightarrow Q(y + 1))))\end{aligned}$$

Représentation des fonctions

Une fonction est représentée par une valeur du type abstrait *Func*. L'opérateur *AppReturns* associe une formule caractéristique à chaque fonction :

$$AppReturns : \forall A, B,\ Func \to A \to (B \to \texttt{Prop}) \to \texttt{Prop}$$

Autrement dit : $AppReturns\, f\ v\ Q$ est la précondition de l'application $f\ v$ avec postcondition Q.

Chaque définition globale $\texttt{let rec}\ f\ x = t$ introduit une nouvelle constante $f : Func$ et un axiome

$$\forall x, Q,\ [\![t]\!]\ Q \Rightarrow AppReturns\, f\ x\ Q$$

Spécifier une fonction

Une spécification de fonction de la forme $\{ P \} \ f \ x \ \{ Q \}$ s'énonce comme un lemme sur *AppReturns f* :

$$\forall x, P\, x \Rightarrow \mathit{AppReturns}\ f\ x\ Q$$

Dans l'exemple précédent :

```
let rec half x =
  if x = 0 then 0 else if x = 1 then fail
  else let y = half (x - 2) in y + 1
```

Voici deux spécifications plausibles :

$$\forall n, n \geq 0 \Rightarrow \mathit{AppReturns}\ f\ (2 \times n)\ (\lambda v.\ v = n)$$
$$\forall n, n \geq 0 \wedge \mathtt{even}(n) \Rightarrow \mathit{AppReturns}\ f\ n\ (\lambda v.\ v = n/2)$$

Spécifier une fonction d'ordre supérieur

Un paramètre f qui est une fonction se spécifie via des hypothèses sur $AppReturns\, f$.

$$\texttt{let}\ \ app\, f = f\ 0$$

Une spécification : ≪si f est à valeurs positives, alors $app\, f$ renvoie un nombre positif≫.

$$\forall f, (\forall x, AppReturns\, f\, x\, (\lambda v.\, v \geq 0)) \Rightarrow AppReturns\, app\, f\, (\lambda v.\, v \geq 0)$$

Une spécification plus précise : ≪ $app\, f$ satisfait toutes les postconditions que $f\ 0$ satisfait≫.

$$\forall f, Q,\ AppReturns\, f\ 0\ Q \Rightarrow AppReturns\ app\ f\ Q$$

Formules caractéristiques pour programmes impératifs

Le système CFML complet traite aussi les programmes ML impératifs (avec des références vers un état mutable).

Préconditions et posconditions utilisent des assertions de la logique de séparation *heap* $\rightarrow$ `Prop` au lieu de propositions `Prop`.

Les formules caractéristiques ne sont plus des calculs de préconditions à partir de postconditions, mais des relations entres préconditions et postconditions :

$$[\![t]\!] : \underbrace{(\mathit{heap} \rightarrow \mathtt{Prop})}_{\text{précondition}} \rightarrow \underbrace{(\lceil \tau \rceil \rightarrow \mathit{heap} \rightarrow \mathtt{Prop})}_{\text{postcondition}} \rightarrow \mathtt{Prop} \quad \text{si } t : \tau$$

F* : types dépendants et monades pour la vérification

Types dépendants, préconditions, postconditions

Dans un langage fonctionnel à types dépendants (comme Agda, Coq, F*), on peut écrire des types qui combinent types de valeurs et propositions logiques :

$\forall x : A.\ P(x) \to B$	fonction prenant un $x : A$ et une preuve de $P(x)$
$\{\, y : B \mid Q(y) \,\}$	paire d'un $y : B$ et d'une preuve de $Q(y)$

Exemple (un type précis pour la fonction ≪racine carrée≫)

$$\forall n : \mathbb{Z},\ n \geq 0 \to \{\, r : \mathbb{Z} \mid r \geq 0 \land r^2 \leq n < (r+1)^2 \}$$

Un type pour les triplets de Hoare

Idée : utiliser des types dépendants pour définir un type $M\ P\ A\ Q$ des calculs c de type A qui satisfont le triplet $\{\, P \,\}\ c\ \{\, Q \,\}$.

Pour des calculs purs, on prend

$$M\,(P : \texttt{Prop})\,(A : \texttt{Type})\,(Q : A \to \texttt{Prop}) : \texttt{Type} := P \to \{\, a : A \mid Q\, a \,\}$$

Ce type forme une monade, avec les opérations

$$\texttt{ret}\ v = \lambda p.\ \langle v, p \rangle$$

$$\texttt{bind}\ m\ f = \lambda p.\ \texttt{let}\ \ \langle v, q \rangle = m\ p\ \ \texttt{in}\ \ f\ x\ q$$

Les monades de Hoare

Le plus intéressant dans ces opérations monadiques, c'est leur type :

$$\texttt{ret} : \forall (A : \texttt{Type})\ (a : A)(Q : A \to Prop),\ M\ (Q\ v)\ A\ Q$$

$$\texttt{bind} : \forall (A\ B\ C : \texttt{Type})\ (P : \texttt{Prop})\ (Q : A \to \texttt{Prop})\ (R : B \to \texttt{Prop}),$$
$$M\ P\ A\ Q \to (\forall x : A, M\ (Q\ x)\ B\ R) \to M\ P\ A\ R$$

Ces types correspondent exactement à des règles de logique de Hoare (dans le style de notre langage PTR) :

$$\{\, Q\ [\![a]\!] \,\}\ a\ \{\, Q \,\} \qquad \frac{\{\, P \,\}\ c\ \{\, Q \,\} \qquad \forall x,\ \{\, Q\ x \,\}\ c'\ \{\, R \,\}}{\{\, P \,\}\ \texttt{let}\ \ x = c\ \ \texttt{in}\ \ c'\ \{\, R \,\}}$$

≪La≫ monade de Hoare : état mutable

(Nanevski *et al*, Hoare Type Theory (2006); Ynot (2008))

Si *State* est le type des états, la monade d'état usuelle est

$ST\ A = State \to A \times State$ (état ≪avant≫ → valeur, état ≪après≫)

La monade de Hoare correspondante est

$$ST\ P\ A\ Q = \forall s : State, P\ s \to \{\, (a, s') \mid Q\ a\ s' \,\}$$

avec $P : State \to \texttt{Prop}$ et $Q : A \to State \to \texttt{Prop}$
(assertions sur les états).

`ret` et `bind` ont leurs types usuels.

Opérations sur l'état mutable

On peut donner des types aux opérations sur l'état mutable qui correspondent aux ≪grandes≫ règles de la logique de séparation :

$$\begin{aligned}
\texttt{get}\ \ell &: \forall v, R, ST\ (\ell \mapsto v \star R)\ \mathrm{Z}\ (\lambda r.\ \langle r = v\rangle \star \ell \mapsto v \star R) \\
\texttt{set}\ \ell\ v &: \forall R, ST\ (\ell \mapsto _ \star R)\ \texttt{unit}\ (\lambda _.\ \ell \mapsto v \star R) \\
\texttt{alloc} &: \forall R, ST\ R\ \texttt{addr}\ (\lambda \ell.\ \ell \mapsto _ \star R) \\
\texttt{free}\ \ell &: \forall R, ST\ (\ell \mapsto _ \star R)\ \texttt{unit}\ (\lambda _.\ R)
\end{aligned}$$

Une monade de séparation

On peut retrouver les «petites» règles et gagner la règle d'encadrement en quantifiant sur tous les encadrements :

$$STsep\ P\ A\ Q = \forall R,\ ST\ (P \star R)\ A\ (\lambda v.\ Q\ v \star R)$$

L'encadrement se présente comme une fonction de retypage :

$$\texttt{frame}\ R : STsep\ P\ A\ Q \to STsep\ (P \star R)\ A\ (\lambda v.\ Q\ v \star R)$$

Les «petites» règles sont bien là :

$$\texttt{ret}\ v : STsep\ \texttt{emp}\ A\ (\lambda r.\ \langle r = v \rangle)$$

$$\texttt{get}\ \ell : \forall v, STsep\ (\ell \mapsto v)\ \texttt{Z}\ (\lambda r.\ \langle r = v \rangle \star \ell \mapsto v)$$

$$\texttt{set}\ \ell\ v : STsep\ (\ell \mapsto _)\ \texttt{unit}\ (\lambda _.\ \ell \mapsto v)$$

$$\texttt{alloc} : STsep\ \texttt{emp}\ \texttt{addr}\ (\lambda \ell.\ \ell \mapsto _)$$

$$\texttt{free}\ \ell : STsep\ (\ell \mapsto _)\ \texttt{unit}\ \texttt{emp}$$

Monade de Hoare relationnelle

Pour mémoire : le système Ynot de Nanevsky *et al* encode une logique de Hoare relationnelle, où la postcondition relie l'état initial et l'état final :

$$STrel\ P\ A\ Q = \forall s,\ P\ s \to \{\, (a, s') \mid Q\ a\ s\ s' \,\}$$

avec $Q : A \to State \to State \to \texttt{Prop}$.

Cela permet d'éviter des variables auxiliaires dans certaines règles, mais complique le typage du `bind` :

$$\begin{aligned}\texttt{bind} :\ & \forall A, B, P_1, Q_1, P_2, Q_2, \\ & STrel\ P_1\ A\ Q_1 \to (\forall (a : A), STrel\ (P_2\ a)\ B\ (Q_2\ a)) \to STrel\ P\ B\ Q\end{aligned}$$

avec $P = \lambda s_1.\ P_1\ s_1 \land \forall a, s_2.\ Q_1\ a\ s_1\ s_2 \Rightarrow P_2\ s_2$
et $Q = \lambda b, s_1, s_3.\ \exists a, s_2.\ Q_1\ a\ s_1\ s_2 \land Q_2\ a\ b\ s_2\ s_3$.

Un bilan sur les monades de Hoare

L'approche ≪programmer et vérifier en même temps≫ des types dépendants, où de plus

- on peut utiliser des effets,
- la programmation suit un style monadique,
- la vérification suit un style de logique de Hoare.

Le plongement dans Coq (système Ynot) est difficile d'emploi :

- peu d'inférence des assertions intermédiaires ;
- besoin de fonctions de retypage qui matérialisent les règles purement logiques (conséquence, encadrement) :

$$\texttt{cons_pre} : (P' \to P) \to ST\ P\ A\ Q \to ST\ P'\ A\ Q$$

L’approche F*

Le langage F* utilise aussi des types dépendants pour programmer et vérifier en présence d’effets, mais avec une approche un peu différente :

- Monades de Dijkstra au lieu de monades de Hoare ($\approx$ calcul de précondition au lieu de triplets).
- Typeur spécifique qui infère des conditions de vérification et les résout automatiquement si possible.
- Hiérarchie d’effets et de monades permettant de traiter chaque partie du code avec le minimum d’effets nécessaires.

Monades de Dijkstra

Idée : pour un calcul c, au lieu de considérer les pré et post conditions P, Q et les triplets $\{\, P \,\}\; c\; \{\, Q \,\}$, on considère les transformateurs de prédicats $W : POST \to PRE$
et les triplets $\{\, W\, Q \,\}\; c\; \{\, Q \,\}$ pour toutes les postconditions Q.

Exemple : la monade d'état.

$$PRE = State \to \mathtt{Prop}$$

$$POST\ A = A \to State \to \mathtt{Prop}$$

$$TRANSF\ A = POST\ A \to PRE$$

$$ST\ A\ (W : TRANSF\ A) = \forall Q, s,\ W\ Q\ s \to \{\, (a, s') \mid Q\ a\ s' \,\}$$

Le type $ST\ A\ W$ est le type des calculs monadiques produisant une valeur de type A et respectant le ≪contrat≫ W.

Les opérations de la monade de Dijkstra pour l'état

$$RET\ (v : A) : TRANSF\ A = \lambda Q.\ Q\ v$$

$$\texttt{ret}\ (v : A) : ST\ A\ (RET\ v) = \lambda Q, s, p, \langle (v, s), p \rangle$$

Pour le `bind`, avec $W_1 : TRANSF\ A$ et $W_2 : A \to TRANSF\ B$ et $m : ST\ A\ W_1$ et $f : \forall a : A, ST\ B\ (W_2\ a)$,

$$BIND\ W_1\ W_2 : TRANSF\ B = \lambda Q.\ W_1\ (\lambda a.\ W_2\ a\ Q)$$

$$\texttt{bind}\ m\ f : ST\ A\ (BIND\ W_1\ W_2) = \ldots$$

Remarque : les types de `ret` et `bind` sont toujours de la forme ci-dessus pour toutes les monades de Dijkstra ; seuls les opérateurs *RET*, *BIND* changent.

Remarque : *RET* et *BIND* forment aussi une monade !

Les opérations de la monade de Dijkstra pour l'état

Les opérations sur la mémoire suivent le même patron :

$$GET\ \ell : TRANSF\ Z = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ (s\ \ell)\ s$$

$$\mathtt{get}\ \ell : ST\ Z\ (GET\ \ell)$$

$$SET\ \ell\ v : TRANSF\ \mathtt{unit} = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ ()\ s[\ell \leftarrow v]$$

$$\mathtt{set}\ \ell\ v : ST\ \mathtt{unit}\ (SET\ \ell)$$

$$ALLOC : TRANSF\ \mathtt{addr} = \lambda Q, s,\ \forall \ell \notin Dom(s), Q\ \ell\ s[\ell \leftarrow 0]$$

$$\mathtt{alloc} : ST\ \mathtt{addr}\ ALLOC$$

$$FREE\ \ell : TRANSF\ \mathtt{unit} = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ ()\ (s \setminus \ell)$$

$$\mathtt{free}\ \ell : ST\ \mathtt{unit}\ (FREE\ \ell)$$

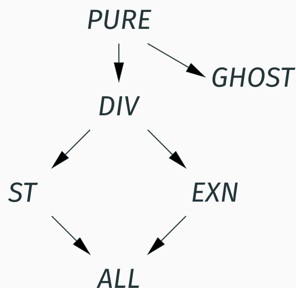
Remarque : on peut définir `get`, `set`, ..., en accord avec notre définition de ST ; mais on peut aussi les laisser abstraits, ce qui donne une axiomatisation d'un effet «mémoire mutable» primitif.

La monade de Dijkstra pour les exceptions

Les postconditions portent sur les deux types de résultats, normaux ou exceptionnels.

$$\begin{aligned}
PRE &= \texttt{Prop} \\
POST\ A &= (A + \texttt{exn}) \to \texttt{Prop} \\
TRANSF\ A &= POST\ A \to PRE \\
EXN\ A\ W &= \forall Q : POST\ A, ,\ W\ Q \to \{\, r \mid Q\ r \,\} \\
RET\ v &= \lambda Q.\ Q\,(\texttt{left}\ \ v) \\
BIND\ W_1\ W_2 &= \lambda Q.\ W_1\,(\lambda r.\,\texttt{match}\ \ r\ \ \texttt{with} \\
&\qquad \mid \texttt{left}\ \ v \Rightarrow W_2\ v\ Q \\
&\qquad \mid \texttt{right}\ \ e \Rightarrow Q\,(\texttt{right}\ \ e))
\end{aligned}$$

Une hiérarchie de monades



Chaque flèche correspond à un transformateur de monades, p.ex.

$$EXN\ A\ W \rightarrow ALL\ A\ (EXN_to_ALL\ W)$$

Inférence de la plus petite monade = inférence d'effets

Les calculs sont automatiquement placés dans la plus petite monade qui leur suffit.

Exemple : la règle du `let` (composition séquentielle).

$$\frac{\Gamma \vdash e_1 : M_1\ \tau_1\ W_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : M_2\ \tau_2\ W_2 \\ M = M_1 \sqcup M_2 \qquad W_1' = M_1_to_M\ W_1 \qquad W_2' = M_2_to_M\ W_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : M\ \tau_2\ (M.BIND\ W_1'\ (\lambda x.\ W_2'))}$$

Point d’étape

Point d’étape

Un bel exemple de logique de programmes pour un langage fonctionnel : F* et ses applications à la vérification de bibliothèques cryptographiques.

D’autres approches sont possibles (p.ex. CFML, Iris).
Pas de consensus.

Les fonctions d’ordre supérieur (`map`, `iter`, `fold`, ...) sont difficiles à spécifier, particulièrement en présence d’état mutable.

Un puzzle

Le *awkward example* de Pitts et Stark :

```
let awkward =
  let r = ref 0 in
  fun f -> assert (!r mod 2 = 0); incr r; f(); incr r
```

L'assertion est fausse si `awkward` est appliquée à elle-même…

Quelle spécification donner à `awkward` ?

Bibliographie

Bibliographie

Le langage F* : `https://www.fstar-lang.org/`

Le système CFML : `https://www.chargueraud.org/softs/cfml/`

Les fonctions comme valeurs de première classe en logique de séparation :

- L. Birkedal, A. Bizjak, *Lecture Notes on Iris : Higher-Order Concurrent Separation Logic*, chapitres 4 à 6.

FIN