Un système de types pragmatique pour la vérification déductive des programmes

Léon Gondelman

Université Paris Saclay, LRI, Inria, CNRS

thèse financée par le projet ANR BWare

thèse présentée le 13 décembre 2016
devant le jury composé de

Mme	Sandrine Blazy	(rapporteure)
M.	Giuseppe Castagna	(examinateur)
M.	Jean Goubault-Larrecq	(examinateur)
M.	Jean-Christophe Filliâtre	(directeur de thèse)
M.	Andrei Paskevich	(directeur de thèse)
M.	Jorge Sousa Pinto	(rapporteur)
M.	François Pottier	(examinateur)

(maman, amis) : C'est quoi le sujet de ta thèse ?

Sur comment utiliser les maths : **(moi)**
pour faire des programmes sans bugs

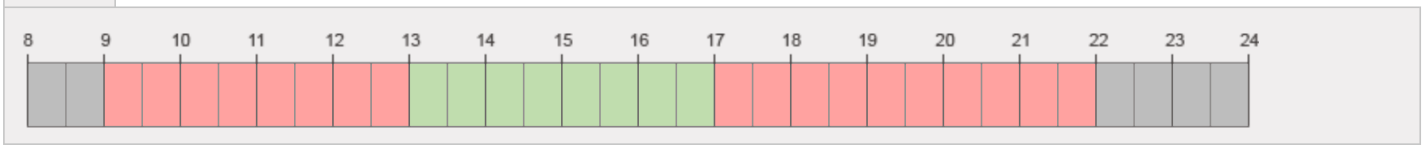
(maman, amis) : C'est quoi le sujet de ta thèse ?

Sur comment utiliser les maths : **(moi)**
pour faire des programmes sans bugs

(maman, amis) : Ah ouais ? Et ça marche donc
de tester les programmes ?

et c'est là que ça prend 45 minutes à expliquer...

4 déc.



Réservation

Les réservations sont a confirmer par le studio.

Date :	dimanche 04 décembre 2016
Heure de début :	20:00
Heure de fin :	11:30
Groupe :	GONDELMANS Léon
Type :	Solo
Commentaires :	
Prix :	**-46.75**

* Le prix est donné à titre indicatif et tient compte des réductions éventuelles du groupe. Il peut être ajusté ou remplacé par des heures de forfait par le studio au moment de la confirmation

Réserver

A bug in fMRI software could invalidate 15 years of brain research

This is huge.

BEC CREW 6 JUL 2016

There could be a very serious problem with the past 15 years of research into human brain activity, with a new study suggesting that a bug in fMRI software could invalidate the results of some 40,000 papers.

That's massive, because functional magnetic resonance imaging (fMRI) is one of the best tools we have to measure brain activity, and if it's flawed, it means all those conclusions about what our brains look like during things like exercise, gaming, love, and drug addiction are wrong.

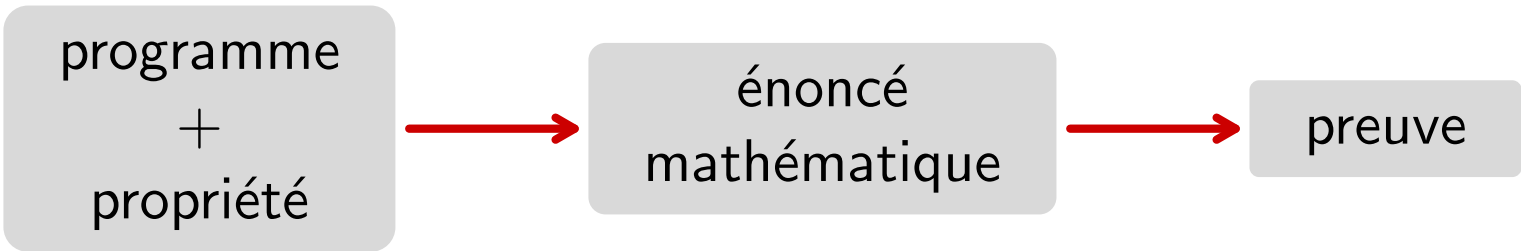
questions de confiance

comment augmenter la confiance envers les logiciels ?

quelques réponses :

- être discipliné
- effectuer des tests
- utiliser des méthodes d'analyse statique
  - typage
  - model checking
  - interprétation abstraite
  - vérification déductive

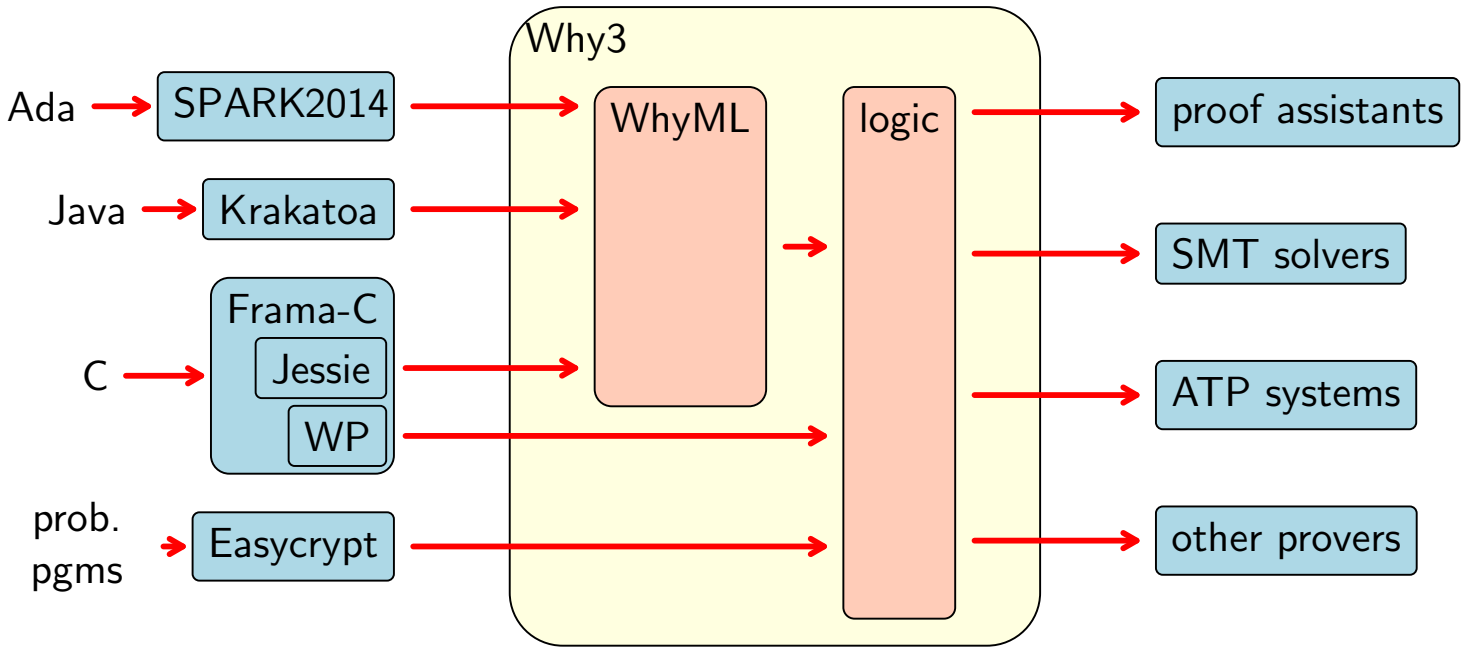
vérification déductive des programmes



1) spécifier le comportement d'un programme
2) générer des conditions de vérification
3) prouver ces conditions avec
   - des démonstrateurs automatiques
   - des assistants de preuve interactifs

outils de vérification : *Dafny*, *KIV*, *Verifast*, *Viper*, *Why3*, ...

Why3


Why3
Ada
SPARK2014
Java
Krakatoa
Frama-C
C
Jessie
WP
prob.
pgms
Easycrypt
WhyML
logic
proof assistants
SMT solvers
ATP systems
other provers

algorithme d'Euclide

```
let gcd (p: int) (q: int) : int
  requires { p ≥ 0 ∧ q ≥ 0 }
  ensures  { result = gcd p q }
= let x = ref p in
  let y = ref q in
  while !y > 0 do
    variant   { !y }
    invariant { !x ≥ 0 ∧ !y ≥ 0 }
    invariant { gcd !x !y = gcd p q }
    let r = mod !x !y in
    x := !y;
    y := r;
  done;
  !x
```

cette thèse

but : poser un cadre formel pour certains aspects de la vérification déductive

focus : utiliser un système de types pour assurer des propriétés de correction

plan :
1) code fantôme
2) contrôle statique des alias
3) raffinement des données

.I.
code fantôme

exemple

```
let rec aux (a b n: int) : int
= if n = 0 then a else aux b (a+b) (n-1)


let fib (n: int) : int
= aux 0 1 n
```

spécification de `fib`

```
let rec aux (a b n: int) : int


= if n = 0 then a else aux b (a+b) (n-1)

let fib (n: int) : int
  requires {0 ≤ n}
  ensures  {result = F_n}
= aux 0 1 n
```

spécification de aux

```
let rec aux (a b n: int) : int
  requires {0 ≤ n}
  requires {∃k. 0 ≤ k ∧ a = F_k ∧ b = F_{k+1}}
  ensures  {∃k. 0 ≤ k ∧ a = F_k ∧ b = F_{k+1} ∧ result = F_{k+n}}
= if n = 0 then a else aux b (a+b) (n-1)

let fib (n: int) : int
  requires {0 ≤ n}
  ensures  {result = F_n}
= aux 0 1 n
```

indice explicite

```
let rec aux (k: int) (a b n: int) : int
  requires {0 ≤ k ∧ 0 ≤ n}
  requires {a = F_k ∧ b = F_{k+1}}
  ensures  {result = F_{k+n}}
= if n = 0 then a else aux (k+1) b (a+b) (n-1)

let fib (n: int) : int
  requires {0 ≤ n}
  ensures  {result = F_n}
= aux 0 0 1 n
```

cependant, ce programme

- n'est pas exactement le programme original
- fait des calculs superflus

effacement

or ces calculs *fantômes*

- n'ont pas d'impact sur le reste du programme
- ne font pas d'effets de bord

on peut remplacer ce code fantôme par une valeur unit :

```
let rec aux () a b n =
  if n = 0 then a else aux () b (a+b) (n-1)

let fib n = aux () 0 1 n
```

effacement

or ces calculs *fantômes*

- n'ont pas d'impact sur le reste du programme
- ne font pas d'effets de bord

ou effacer ce code fantôme et obtenir le programme initial

```
let rec aux a b n =
  if n = 0 then a else aux b (a+b) (n-1)

let fib n = aux 0 1 n
```

l'esprit du code fantôme

le code fantôme facilite la spécification/preuve

il doit respecter le principe de non-interférence :

> *le code fantôme peut être effacé du programme sans aucune modification dans le comportement observable du programme*

en d'autres termes,

- le code du programme n'accède pas aux données fantômes
- le code fantôme ne modifie pas les données du programme
- le code fantôme n'altère pas le flot de contrôle et termine

le code fantôme est utilisé dans de nombreux outils de vérification

Dafny, *VCC*, *VeriFast*, *Why3*

on y trouve notamment

- des fonctions avec des paramètres fantômes
- des fonctions fantômes
- des variables fantômes et champs fantômes (modèles)

dans tous les cas, la non-interférence doit être garantie

dans une bibliothèque externe :

```
type nat = O | S of nat

let rec plus (x y: nat) = match x with
  | O    → y
  | S x' → S (plus x' y)
```

programme avec code fantôme :

```
let rec aux (ghost k: nat) (a b n: nat) = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'

let fib n = aux O O (S O) n
```

effets de bord 1/3

interférence via les effets d'écriture :

```
val countdown: ref int

let rec plus (x y: nat) = match x with
  | O    →  y
  | S x' →
      decr countdown;
      S (plus x' y)
```

```
let rec aux (ghost k: nat) (a b n: nat) = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

effets de bord 2/3

interférence via les modifications du flot de contrôle :

```
val ghost countdown: ref int

let rec plus (x y: nat) = match x with
  | O    →
      if !countdown = 0
      then raise Zombies;
      y
  | S x' →
      decr countdown;
      S (plus x' y)
```

```
let rec aux (ghost k: nat) (a b n: nat) = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

effets de bord 3/3

interférence via la non terminaison :

```
val ghost countdown: ref int

let rec plus (x y: nat) = match x with
  | O    →
      if !countdown = 0
      then while true do () done
      y
  | S x' →
      decr countdown;
      S (plus x' y)
```

```
let rec aux (ghost k: nat) (a b n: nat) = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

notre approche

quoi : un système de types

- avec un minimum d'annotations du code fantôme
- qui permet la réutilisation
- qui garantit la non-interférence

comment : inférer les effets et statuts fantômes

formalisme

- GhostML = un langage simple à la ML
  - état mutable global
  - fonctions récursives
  - code fantôme
  - sémantique opérationnelle déterministe à petits pas
- effacement du code fantôme $\mathcal{E}$: GhostML $\rightarrow$ MiniML
  - transforme les données fantômes en des valeurs unit
  - préserve les données du programme et leur structure

système de types

typage avec deux indicateurs booléens : $\beta,\ \epsilon \in \{\top, \bot\}$

- β : statut fantôme
- ϵ : effets observables

$$\boxed{\Gamma \vdash e : \tau \cdot \beta \cdot \epsilon}$$

effets latents et statut fantôme du paramètre explicites :

$$\boxed{\Gamma \vdash e : (\tau_1^{\beta_1} \stackrel{\epsilon_1}{\Longrightarrow} \tau) \cdot \beta \cdot \epsilon}$$

dans chaque règle du typage, la condition de non-interférence :

$$\boxed{\beta = \top \implies \epsilon = \bot}$$

les expressions fantômes ne peuvent pas avoir des effets observables

typage de l'application

deux cas distincts pour typer une application $(e\ e_1)$:

- le paramètre formel n'est pas fantôme $\boxed{e : \tau_1^{\bot} \stackrel{\epsilon_1}{\Longrightarrow} \tau}$

  appliquer une fonction à un argument fantôme rend l'application fantôme toute entière...

- le paramètre formel est fantôme $\boxed{e : \tau_1^{\top} \stackrel{\epsilon_1}{\Longrightarrow} \tau}$

  ...sauf si la fonction attend un argument fantôme

théorème de correction :

L'évaluation des programmes typés est préservée par l'opération d'effacement.

formellement :

Soit une expression e bien typée telle que

$$\Gamma \vdash e : \tau \cdot \bot \cdot \epsilon.$$

Alors, les deux diagrammes suivants commutent :

$$\begin{array}{ccc} \mu \cdot e & \longrightarrow^\star & \mu' \cdot v \\ \Downarrow & & \Downarrow \\ \mathcal{E}(\mu) \cdot \mathcal{E}(e) & \longrightarrow^\star & \mathcal{E}(\mu') \cdot \mathcal{E}(v) \end{array} \qquad \begin{array}{ccc} \mu \cdot e & \Longrightarrow & \infty \\ \Downarrow & & \Downarrow \\ \mathcal{E}(\mu) \cdot \mathcal{E}(e) & \Longrightarrow & \infty \end{array}$$

.II.
contrôle statique des alias

aliasing

un concept puissant mais potentiellement dangereux

approches existantes :

- régions [Tofte et Talpin 1997]
- *ownership types* [Clarke 1998]
- permissions, capacités [Crary 1999, Boyland 2001]

preuve de programmes avec pointeurs

nécessite parfois un raisonnement sur le contenu de la mémoire

- plusieurs approches existent :
  - modèles mémoire explicites [Burstall 1972]
  - logique de séparation [Reynolds 2002]
  - *dynamic frames* [Kassios 2006]
- néanmoins, on observe que souvent la preuve peut se faire dans le cadre de la logique de Hoare :
  - ne nécessite pas de modèle mémoire explicite
  - se ramène à des conditions de vérification plus simples

notre approche

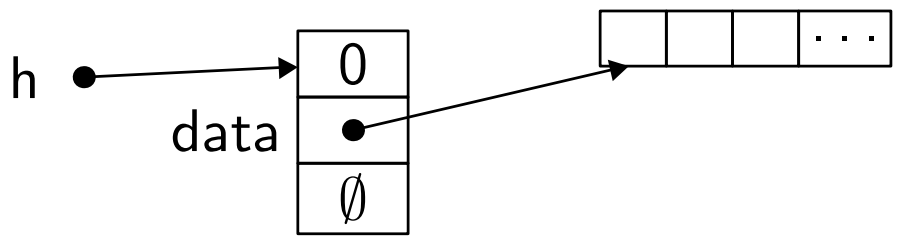
quoi : un contrôle statique des alias *antérieur* à la génération de conditions de vérification

méthode : un système de types avec régions singletons et effets

cadre : structures de données avec des composantes mutables imbriquées d'une profondeur bornée

table de hachage avec adressage ouvert

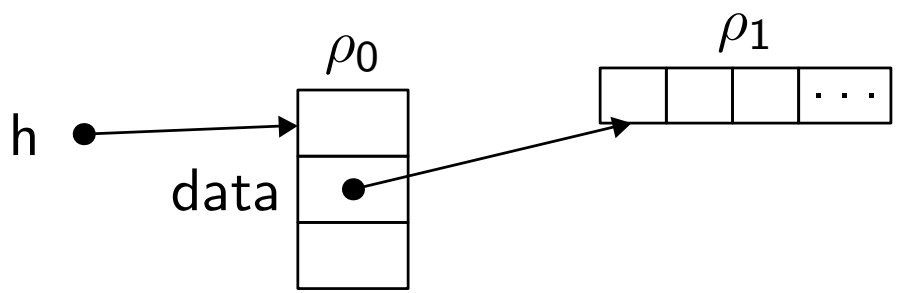
```
type t = { mutable size: int;
           mutable data: array elt;
    ghost mutable view: set elt }
```



régions

l'identité d'une valeur mutable est encodée dans son type,
pas dans son nom

exemple



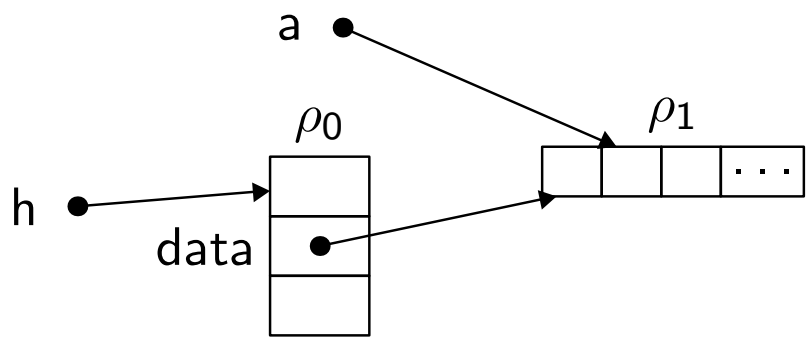
région : un type singleton

invariant : deux noms sont des alias $\iff$ ils ont la même région

alias

```
let a = h.data
```

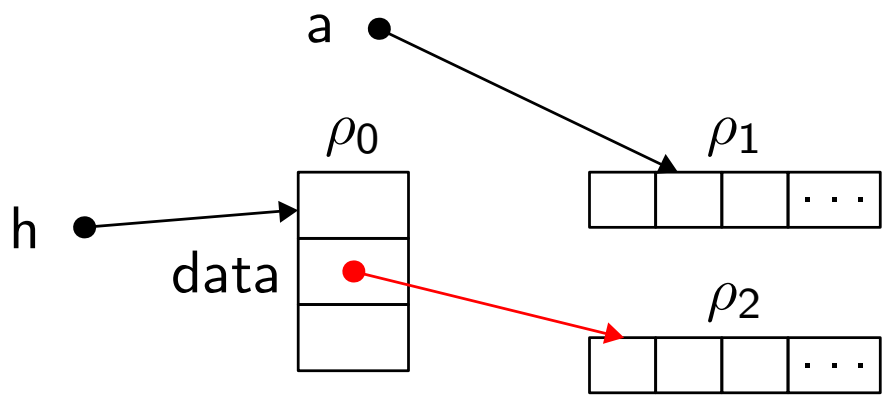
a et h.data ont le même type ρ_1
car représentent la même adresse



affectation 1/2

```
h.data ← newArray 17
```

une possibilité serait de changer le type de `h`
(*strong update* [Berdine 2006])



affectation 2/2

mais que dire de l'affection sous une conditionnelle ?
en général, on ne connaît pas statiquement le résultat du test

```
let a = h.data in
if ... then
  h.data ← newArray 17
```

une solution serait d'utiliser des types dépendants

notre solution

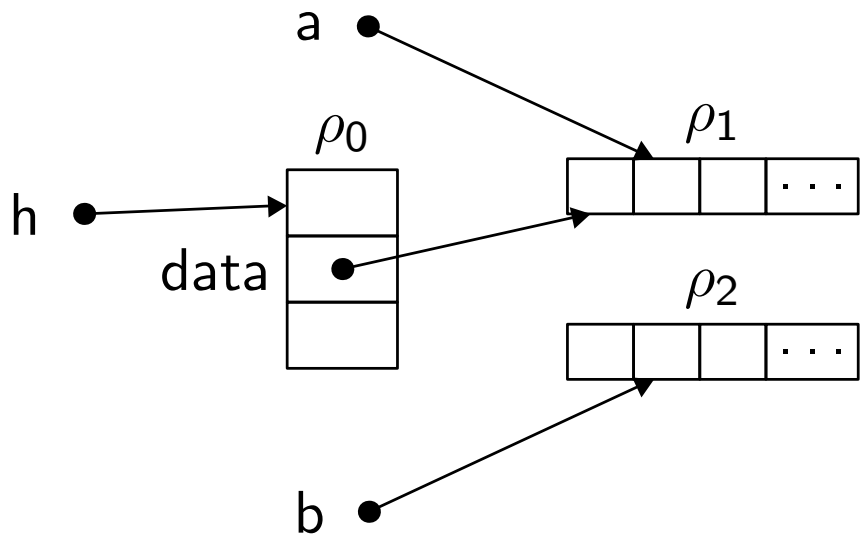
d'abord, on détecte les conflits d'aliasing :

- soit deux noms du même type peuvent ne plus être aliasés,
- soit deux noms de types distincts peuvent devenir aliasés

et on invalide l'utilisation de l'un de ces deux noms dans la suite du programme

conflits d'aliasing

```
let a = h.data in
let b = newArray 17 in
h.data ← b
```



deux conflits d'aliasing :

- `a` et `h.data` ont le même type, mais ne sont plus aliasés
- `b` et `h.data` ont des types distincts mais deviennent aliasés

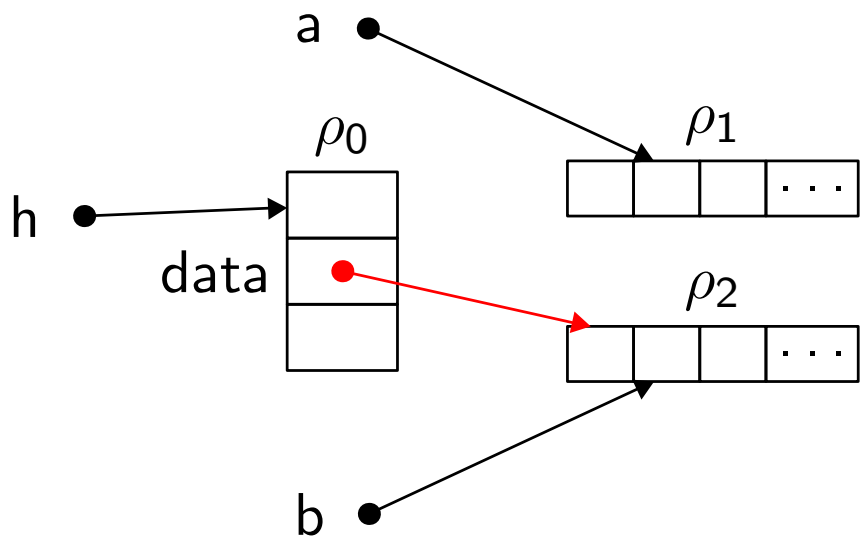
pas de témoin, pas de crime

le système de types invalide les variables `a` et `b` dans la suite du programme après l'affectation

néanmoins, on peut toujours continuer à utiliser `h.data` et on n'a plus besoin de modifier le type de `h.data`

l'invariant de représentation est préservé

```
let a = h.data in
let b = newArray 17 in
h.data ← b
```



effet : *writes* ρ_0 *resets* ρ_1, ρ_2

interdit d'utiliser dans la suite toute variable x existante telle que ρ_1 ou ρ_2 est atteignable depuis le type de x sans passer par la région ρ_0

ainsi, a et b sont invalidées, alors que h ne l'est pas

formalisme

un petit langage avec

- des régions imbriquées de profondeur bornée
- une sémantique opérationnelle déterministe à petits pas
- un contrôle statique des alias

système de types

τ	$::=$	$\nu \mid \rho$	types
ν	$::=$	$\texttt{bool} \mid \texttt{int} \mid \ldots$	types scalaires
ρ	$::=$	$\{ f : \tau, \ldots, f : \tau \}_r$	régions

$$\Gamma \vdash e : \tau \cdot \varepsilon \qquad \varepsilon \triangleq (\omega \cdot \varphi) \qquad \omega \cap \varphi = \varnothing$$

ω = ensemble de régions écrites

φ = ensemble de régions invalidées

au typage, les effets sont

- calculés lors de l'opération d'affectation
- combinés au branchement ou à la mise en séquence
- utilisés pour vérifier la mise en séquence

mise en séquence

$$\frac{\Gamma \vdash e_1 : \tau_1 \cdot \varepsilon_1 \qquad \Gamma \vdash e_2 : \tau_2 \cdot \varepsilon_2 \qquad \varepsilon_1 \triangleright \Gamma(\mathcal{F}_v(e_2))}{\Gamma \vdash e_1;\ e_2 : \tau_2 \cdot \varepsilon_1 \sqcup \varepsilon_2}$$

$\varepsilon_1 \triangleright \Gamma(\mathcal{F}_v(e_2))$ *: toute variable libre dans* e_2 *a un type valide vis-à-vis de l'effet de* e_1

$\varepsilon_1 \sqcup \varepsilon_2$ *: union des effets* ε_1 *et* ε_2

$$(\omega \cdot \varphi) \rhd \tau$$

$\ll \tau$ est valide vis-à-vis de $(\omega \cdot \varphi) \gg$

$$\frac{}{(\omega \cdot \varphi) \rhd \nu} \qquad \frac{\rho \in \omega}{(\omega \cdot \varphi) \rhd \rho}$$

$$\frac{\rho \notin \omega \quad \rho \notin \varphi \quad \forall i.\, (\omega \cdot \varphi) \rhd \rho.f_i}{(\omega \cdot \varphi) \rhd \rho}$$

tout chemin depuis τ vers les régions de φ
passe par une région de ω

union d'effets

Soient deux effets $\varepsilon_1 = (\omega_1 \cdot \varphi_1)$ et $\varepsilon_2 = (\omega_2 \cdot \varphi_2)$.

Définition (Union d'effets)

$$\varepsilon_1 \sqcup \varepsilon_2 \triangleq \left(\{\rho \in \omega_1 \mid \varepsilon_2 \triangleright \rho\} \cup \{\rho \in \omega_2 \mid \varepsilon_1 \triangleright \rho\} \;\cdot\; \varphi_1 \cup \varphi_2\right)$$

Lemme (caractérisation)

$$\forall \varepsilon_1, \varepsilon_2, \tau.\ \varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau \iff (\varepsilon_1 \triangleright \tau \wedge \varepsilon_2 \triangleright \tau)$$

conditionnelle

$$\frac{\Gamma \vdash e_0 : \texttt{Bool} \cdot \bot \qquad \Gamma \vdash e_1 : \tau \cdot \varepsilon_1 \qquad \Gamma \vdash e_2 : \tau \cdot \varepsilon_2}{\Gamma \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \tau \cdot \varepsilon_1 \sqcup \varepsilon_2}$$

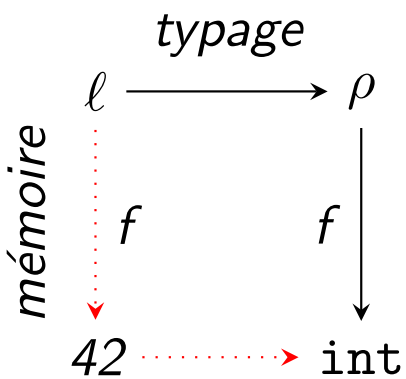
Théorème (préservation)

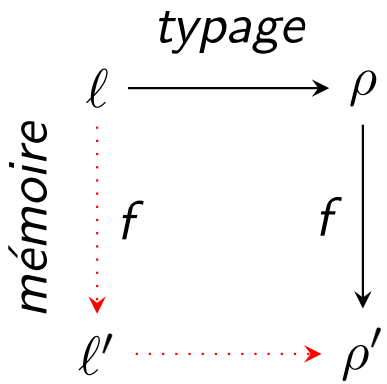
Chaque pas d'exécution

$$\mu \cdot e \longrightarrow \mu' \cdot e'$$

préserve

- *le type de l'expression*
- *la cohérence du typage avec la mémoire :*
  - *des adresses différentes ont des régions différentes*
  - *le typage et la mémoire commutent*



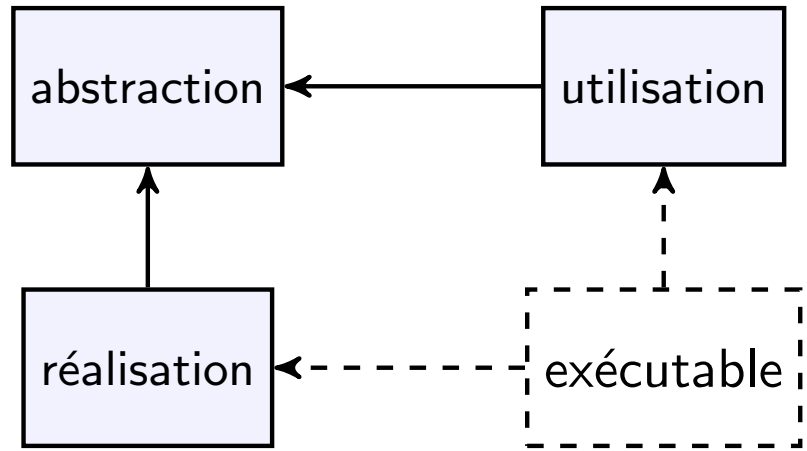


.III.
raffinement des données

abstraction

outil puissant

- du développement modulaire
- de la vérification



l'abstraction permet de

- modéliser des structures de données complexes
- cacher au code client la complexité de la réalisation
- vérifier une grande partie du code dans le cadre proposé

raffinement des données

l'abstraction :

```
type t = private { ghost mutable view: set elt }
```

la réalisation :

```
type t  = { mutable size: int;
            mutable data: array elt;
      ghost mutable view: set elt }
invariant { size = Set.cardinal view ∧ ... }
```

le code client :

```
let rec dfs (x: elt) (marked: t) (ghost on_stack: t): unit
= ...
  add x marked;
  add x on_stack;
  ...
```

le raffinement est exploré dans diverses approches :

- le raffinement dans la méthode B [Abrial 1996]
- *Java Modeling Language* [Leavens et *al.* 1999]
- système des modules dans Dafny [Leino 2014]

conditions nécessaires pour que le raffinement soit correct :

- mécanismes d'encapsulation
- invariants de liaison
- préservation du sens des contrats des fonctions

préservation du typage

```
let rec dfs (x: elt) (marked: t) (ghost on_stack: t): unit
= ...
  add x marked;
  add x on_stack;
  ...
```

- `dfs` suppose qu'il n'y a pas d'alias entre `marked` et `on_stack`
- le raffinement introduit de nouvelles composantes mutables

le raffinement doit préserver le contrôle statique des alias

dans ce travail

objectif : montrer que le contrôle statique des alias est préservé par le raffinement

comment : étendre le formalisme précédent avec les notions de régions privées et de raffinement

régions privées

τ	$::=$	$\nu \mid \rho$	types
ν	$::=$	$\texttt{bool} \mid \texttt{int} \mid \ldots$	types scalaires
ρ	$::=$		régions
	$\mid$	$\{f : \tau, \ldots, f : \tau\}_r$	région publique
	$\mid$	$[f : \nu, \ldots, f : \nu]_r$	région privée

raffinement

$$\nu \quad \overset{\Psi}{\longmapsto} \quad \nu$$

$$\{\, \overrightarrow{f} : \overrightarrow{\tau} \,\}_r \quad \overset{\Psi}{\longmapsto} \quad \{\, \overrightarrow{f} : \overrightarrow{\Psi(\tau)} \,\}_r$$

$$[\, \overrightarrow{f} : \overrightarrow{\nu} \,]_r \quad \overset{\Psi}{\longmapsto} \quad \{\, \overrightarrow{f} : \overrightarrow{\nu}, \overrightarrow{f'} : \overrightarrow{\tau'} \,\}_r$$

---

fraîcheur : tous les champs et les indices introduits sont frais

séparation : les images par le raffinement de deux régions privées distinctes n'ont pas de sous-régions en commun

uniformité : le raffinement préserve l'équivalence structurelle des régions privées

préservation du typage

Théorème

Soit un jugement de typage dérivable

$$\Gamma \vdash e : \tau \cdot \varepsilon.$$

Alors il existe un effet ε' tel que

1) *le jugement $\Psi(\Gamma) \vdash e : \Psi(\tau) \cdot \varepsilon'$ est dérivable*
2) *quel que soit un type $\tau \in dom\ \Psi$, si $\varepsilon \triangleright \tau$, alors $\varepsilon' \triangleright \Psi(\tau)$.*

conclusion et perspectives

un tel système de types est un compromis pragmatique entre

- l'expressivité des données mutables
- la simplicité des conditions de preuve

les perspectives :

- rendre le contrôle statique des alias moins restrictif
- conjuguer notre approche avec des approches existantes