La vérification de bytecode

April 5, 2013

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Math = Zorro

- Quand le test ne suffit pas (trop de cas, criticité)
- Quand le model checking ne marche pas
- Il reste les maths
- Mais les matheux font des erreurs aussi
- Vérification mécanique et sûre des maths

Exemple

- C: compilateur de $L_1 \rightarrow L_2$
- Correct?
- Pour tout programme p, $C(p)$ *a la même sémantique que* p
- Ensemble de test?
- Correction d'un résultat de test? Par un programme? Correct?

Exemple (suite)

- Définir *mathématiquement* les sémantiques de S_1 et S_2 de L_1 et L_2
- Par des spécialistes
- *Démontrer* que $\forall p, S_1(p) = S_2(C(p))$
- Description mathématique de C aussi
- *Fonction* mathématique

Définitions

- Expression: `1`, `1+3`, `true`, `x`, `x or y`...
- $E_I ::= Var(i) \mid i \mid E_I + E_I \mid E_I - E_I \mid E_I$ `or` E_I
- $E_B ::= Var(i) \mid true \mid false \mid E_B$ `or` $E_B \mid\mid E_I > E_I \ldots$

- Instruction:
- $I ::= Var(i) := E \mid$ `if` E `then` I `else` $I \mid I ; I \mid$ *begin* I *end*

- Environnement:
- $\Gamma : \mathbb{N}(variable) \rightarrow \mathbb{N}(valeurs)$
- $\Gamma[i \leftarrow v](j) =$ `if` $j = i$ `then` v `else` $\Gamma(j)$

Sémantique des expressions

$$\frac{< \sigma, E_1 > \mapsto\ n_1 \in \mathbb{N} \qquad < \sigma, E_2 > \mapsto\ n_2 \in \mathbb{N}}{< \sigma, E_1 + E_2 > \mapsto\ n_1 + n_2}$$

$$\frac{\sigma(i) = v}{< \sigma, Var(i) > \mapsto\ v}$$

Sémantique des instruction

$$\frac{< \sigma, E > \mapsto \ n_1 \in \mathbb{N}}{< \sigma, Var(i) := E > \rightsquigarrow \sigma[i \leftarrow n_1]}$$

$$\frac{< \sigma, i_1 > \rightsquigarrow \ \sigma' \qquad < \sigma', i_2 > \rightsquigarrow \ \sigma''}{< i_1 \ ; \ i_2, \sigma > \rightsquigarrow \sigma''}$$

Sémantique des instruction

$$\frac{< \sigma, E > \mapsto \mathit{true} \qquad < \sigma, i_1 > \rightsquigarrow \sigma}{\texttt{if } E \texttt{ then } i_1 \texttt{ else } i_2 \ \rightsquigarrow \ \sigma}$$

$$\frac{< \sigma, E > \mapsto \mathit{false} \qquad < \sigma, i_2 > \rightsquigarrow \sigma}{\texttt{if } E \texttt{ then } i_1 \texttt{ else } i_2 \ \rightsquigarrow \ \sigma}$$

Compcert

- Preuve de correction d'un compilateur C
- Pour powerPC, ARM et x86
- Optimisant
- En Coq

Compcert

- Définition en Coq de:
  - Sémantique de C
  - Sémantique de l'ass. powerPC
  - 7 langages intermédiaires
- Preuve:
  - Préservation de la sémantique pour les 6 traductions
  - Sémantique = valeur des variables + output (*trace*)
- Extraction du compilateur (`ocaml`)
- Compilateur correct *par construction*

Coq

- Assistant de preuve
- Permet les *définitions inductives* et les *preuves par induction*.
  - Types de données inductifs
  - *Propriétés* inductives

```
Inductive nat : Type :=  O : nat | S : nat → nat

⇝ nat: Type.

⇝ nat_ind:
        ∀ P,(P(0) ∧ (∀ n, P(n) → P(S n))) → ∀ n, P(n)
```

Coq

```
Inductive le n: nat →  Prop := (* noté <= *)
  le_n : n <= n
| le_S : ∀ m : nat, n <= m → n <= S m

⇝ le: nat → nat → Prop

⇝ le_ind: ∀ n P,
  P n →
  (∀ m, n <= m ∧ P m → P (S m)) →
  ∀ n', n <= n' → P n'
```

Retour à Compcert

- Typage supposé OK
- Expressions accompagnées de leur type

Compcert – Définitions

```
Inductive expr : Type :=
| Econst_int: int → type → expr
| Econst_float: float → type → expr
| Evar: ident → type → expr
...
Inductive statmnt : Type :=
  (* assignment [lvalue = rvalue] *)
| Sassign: expr → expr → statmnt
  (* sequence *)
| Ssequence: statmnt → statmnt → statmnt
  (* conditional *)
| Sifthenelse: expr → statmnt → statmnt → statmnt
...
```

Compcert – Sémantique des expressions

```
Inductive eval_expr: expr → val → Prop :=

| eval_Etempvar:  ∀ id ty v,
    Env(id) = v → eval_expr (Etempvar id ty) v

...
```

$$\frac{\sigma(i) = v}{< \sigma, Var(i) > \mapsto v}$$

Compcert – Sémantique des expressions

```
...
| eval_Ebinop: ∀ op E1 E2 ty v1 v2 v,
    eval_expr E1 v1 →   (* < σ, E1 > ↦  v1  *)
    eval_expr E2 v2 →   (* < σ, E2 > ↦  v2  *)
    sem_binary_operation op v1
      (typeof E1) v2 (typeof E2) Env = Some v →
    eval_expr (Ebinop op E1 E2 ty) v (*< σ, E1 op E2 >↦ v *)
...
```

$$\frac{< \sigma, E_1 > \mapsto v_1 \in \mathbb{N} \qquad < \sigma, E_2 > \mapsto v_2 \in \mathbb{N}}{< \sigma, E_1 + E_2 > \mapsto v_1 + v_2}$$

Compcert – Sémantique des instructions

```
Inductive step: state → state → Prop :=
| step_assign:   ∀ Var E k m v m',
    eval_expr e le m E v →
    store_value_of_type m Var v = Some m' →
    step (State (Sassign Var E_2) k e m)
         (State Sskip m')
```

$$\frac{< \sigma, E > \mapsto \ n_1 \in \mathbb{N}}{< \sigma, Var(i) := E > \rightsquigarrow \sigma[i \leftarrow n_1]}$$

Compcert – Sémantique des instructions

```
| step_seq:  ∀ f s1 s2 k e le m,
    step (State f (Ssequence s1 s2) k e le m)
         (State f s1 (Kseq s2 k) e le m)
```

$$\frac{< \sigma, i_1 > \rightsquigarrow \sigma' \qquad < \sigma', i_2 > \rightsquigarrow \sigma''}{< i_1 \ ; \ i_2, \sigma > \rightsquigarrow \sigma''}$$

Compcert – Sémantique des instructions

```
| step_ifthenelse_true:  ∀ f a s1 s2 k e le m v_1,
    eval_expr e le m a v_1 →
    is_true v_1 (typeof a) →
    step (State f (Sifthenelse a s1 s2) k e le m)
         (State f s1 k e le m)
| step_ifthenelse_false: ∀ f a s1 s2 k e le m v_1,
...
```

$$\frac{< \sigma, E > \mapsto \mathit{true} \qquad < \sigma, i_1 > \rightsquigarrow \sigma}{\texttt{if } E \texttt{ then } i_1 \texttt{ else } i_2 \ \rightsquigarrow \ \sigma}$$

JavaCard

- Téléchargement d'applets potentiellement malicieuses
- Différents mécanismes de sécurité
- L'un d'eux: *Vérification de bytecode*
- Analyse *statique* du bytecode d'une applet
- Différent de l'analyse statique à partir du code source
- Même problématique pour les navigateurs web

Vérification

- Applets exécutées par une *machine virtuelle* embarquée
- Vérification d'une applet *par rapport* à une V.M. donnée
  1. Définir la sémantique de la V.M.
  2. Analyse statique du bytecode par rapport à la sémantique
  3. Propriété : Si la vérification accepte le bytecode, alors un certain type d'erreur ne se produira jamais à l'exécution

Analyse

Une certaine classe d'erreurs ne se produira *jamais à l'exécution*:

- Typage (pile + registres)
- Dépassements de pile (stack over/underflow)
- Initialisation des registres
- Initialisation des objets

Nombre de configurations trop grand pour être parcouru
$\Rightarrow$ comment s'assurer de ces propriétés avec un nombre raisonnable de vérifications?

- Remplacer les valeurs par les types
- Approximation (abstraction) de la machine virtuelle

Analyse

Une certaine classe d'erreurs ne se produira *jamais à l'exécution*:

- Typage (pile + registres)
- Dépassements de pile (stack over/underflow)
- Initialisation des registres
- Initialisation des objets

Nombre de configurations trop grand pour être parcouru
$\Rightarrow$ comment s'assurer de ces propriétés avec un nombre raisonnable de vérifications?

- Remplacer les valeurs par les types
- Approximation (abstraction) de la machine virtuelle

Exemple de bytecode

```
static int factorielle(int n) {
  int res;
  for (res = 1; n > 0; n--) res = res * n;
  return res; }
```

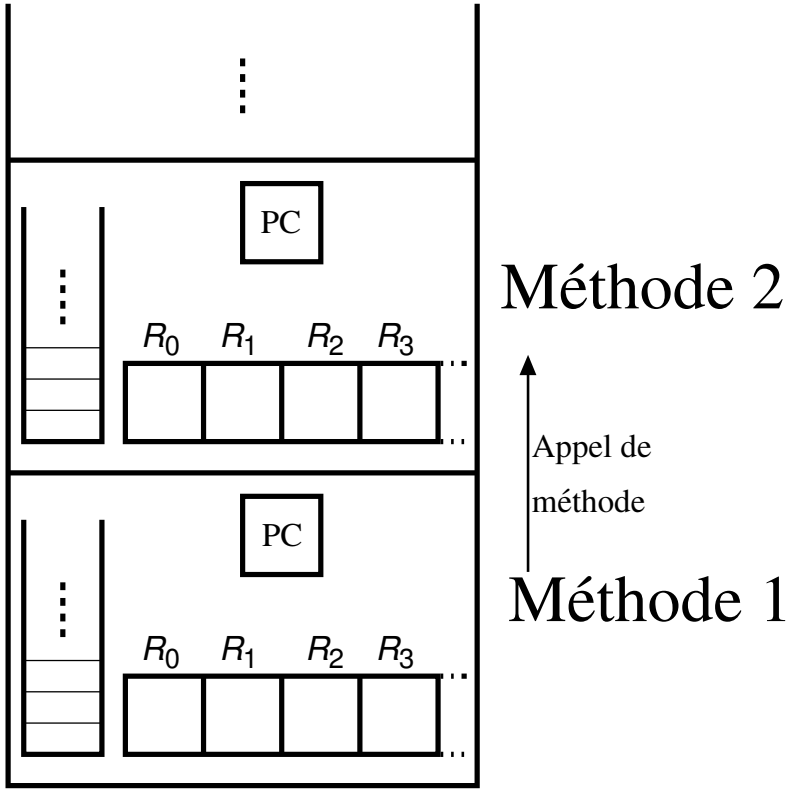
Bytecode JVM correspondant:

```
method static int factorielle(int), 2 registers
0: iconst_1 // push the integer constant 1
1: istore_1 // store it in register 1 (the res variable)
2: iload_0 // push register 0 (the n parameter)
3: ifle 14 // if negative or null, go to PC 14
6: iload_1 // push register 1 (res)
7: iload_0 // push register 0 (n)
8: imul // multiply the two integers at top of stack
9: istore_1 // pop result and store it in register 1
10: iinc 0, -1 // decrement register 0 (n) by 1
11: goto 2 // go to PC 2
14: iload_1 // load register 1 (res)
15: ireturn // return its value to caller
```

La pile d'exécution

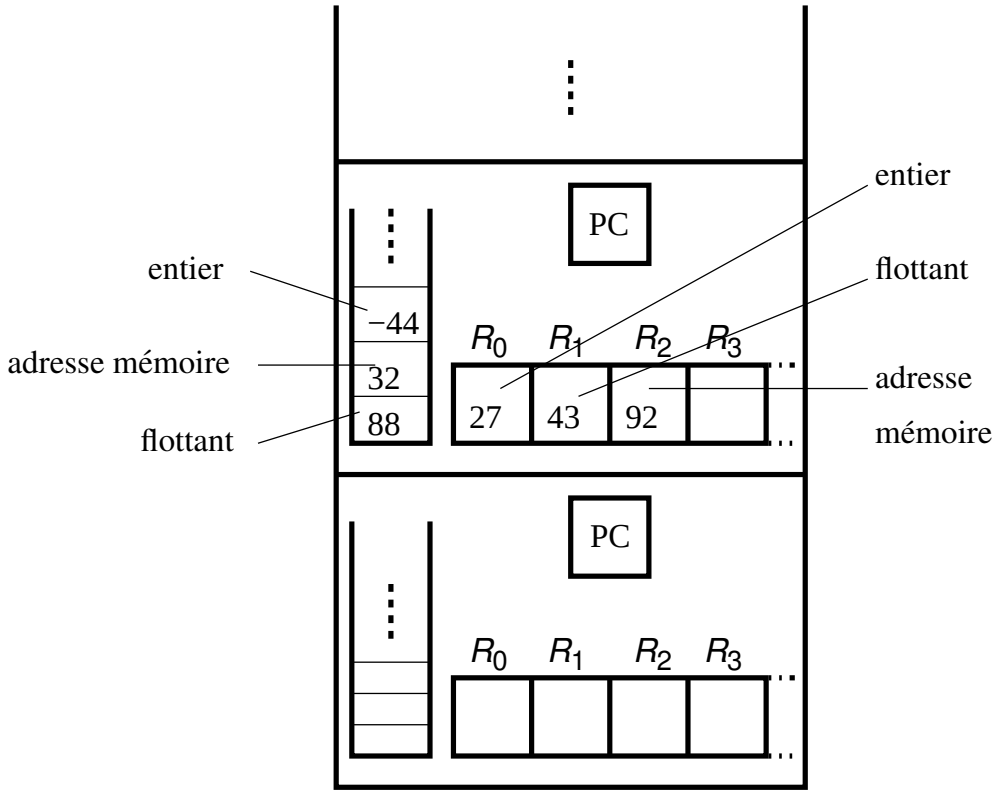
Représente l'*état* de la machine virtuelle.
Pour chaque méthode appelée:

- Une pile d'opérandes
- Des registres (arguments + variables locales)
- Un compteur programme
- Des rattrapeurs (*handlers*) d'exceptions...



Machine offensive

- Embarquée sur la carte, exécute les applets
- Manipule les données *brutes*
- Pas d'information sur les valeurs
- Pas de vérification à l'exécution
- Opérations typées malgré tout



Exemple Offensif

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)	
	$\emptyset$	12	138	args dans les registres initial^t
* 0: Iload 0	12	12	138	
* 1: ifle 4	$\emptyset$	12	138	
*2: Iload 0	12	12	138	
*3: Istore 1	$\emptyset$	12	12	
*4: Aload 1	12	12	12	

Adresse «forgée», ne devrait pas être utilisée, pas détectée.

Exemple Offensif

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	12	138
* 0: Iload 0	12	12	138
* 1: ifle 4	$\emptyset$	12	138
* 2: Iload 0	12	12	138
* 3: Istore 1	$\emptyset$	12	12
* 4: Aload 1	12	12	12

Adresse «forgée», ne devrait pas être utilisée, pas détectée.

Exemple Offensif

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	12	138
* 0: Iload 0	12	12	138
* 1: ifle 4	$\emptyset$	12	138
*2: Iload 0	12	12	138
*3: Istore 1	$\emptyset$	12	12
*4: Aload 1	12	12	12

Adresse «forgée», ne devrait pas être utilisée, pas détectée.

Exemple Offensif

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	12	138
* 0: Iload 0	12	12	138
* 1: ifle 4	$\emptyset$	12	138
*2: Iload 0	12	12	138
*3: Istore 1	$\emptyset$	12	12
*4: Aload 1	12	12	12

Adresse «forgée», ne devrait pas être utilisée, pas détectée.

Exemple Offensif

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	12	138
* 0: Iload 0	12	12	138
* 1: ifle 4	$\emptyset$	12	138
*2: Iload 0	12	12	138
*3: Istore 1	$\emptyset$	12	12
*4: Aload 1	12	12	12

Adresse «forgée», ne devrait pas être utilisée, pas détectée.

Exemple Offensif

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	12	138
* 0: Iload 0	12	12	138
* 1: ifle 4	$\emptyset$	12	138
*2: Iload 0	12	12	138
*3: Istore 1	$\emptyset$	12	12
*4: Aload 1	12	12	12

Adresse «forgée», ne devrait pas être utilisée, pas détectée.

Vérifications

- Typage (pile + registres)
- Dépassements de pile (stack over/underflow)
- Initialisation des registres
- Initialisation des objets

⇒ Informations supplémentaires sur les valeurs (type, initialisation...)
Machine défensive

Vérifications

- Typage (pile + registres)
- Dépassements de pile (stack over/underflow)
- Initialisation des registres
- Initialisation des objets

$\Rightarrow$ Informations supplémentaires sur les valeurs (type, initialisation...)

Machine défensive

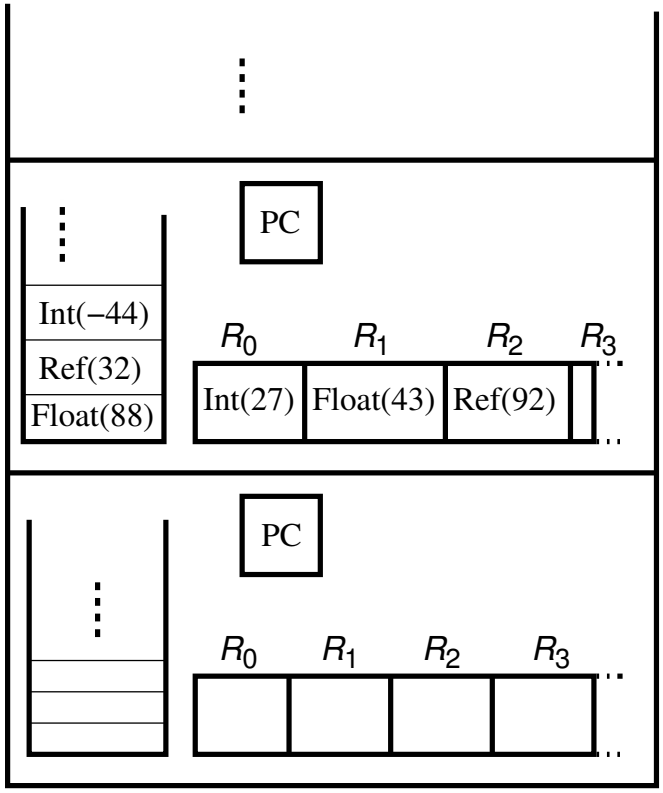
Vérifications

- Typage (pile + registres)
- Dépassements de pile (stack over/underflow)
- Initialisation des registres
- Initialisation des objets

$\Rightarrow$ Informations supplémentaires sur les valeurs (type, initialisation...)
Machine défensive

Machine défensive

- Pas sur la carte
- Manipule les données *typées*
- Informations sur les valeurs
- Vérification à l'exécution des propriétés voulues



Exemple Défensif 1

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	`Int(12)`	`Ref(138)`
* `0: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `1: ifle 4`	$\emptyset$	`Int(12)`	`Ref(138)`
* `2: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `3: Istore 1`	$\emptyset$	`Int(12)`	`Int(12)`
* `4: aload 1`	Bug	`Int(12)`	`Int(12)`

Adresse ≪forgée≫ détectée, arrêt de l'exécution

Exemple Défensif 1

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

		stack	r_0	r_1(adresse)
		∅	`Int(12)`	`Ref(138)`
*	`0: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
*	`1: ifle 4`	∅	`Int(12)`	`Ref(138)`
*	`2: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
*	`3: Istore 1`	∅	`Int(12)`	`Int(12)`
*	`4: aload 1`	Bug	`Int(12)`	`Int(12)`

Adresse ≪forgée≫ détectée, arrêt de l'exécution

Exemple Défensif 1

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int(12)`	`Ref(138)`
* `0: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `1: ifle 4`	∅	`Int(12)`	`Ref(138)`
*`2: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
*`3: Istore 1`	∅	`Int(12)`	`Int(12)`
*`4: aload 1`	Bug	`Int(12)`	`Int(12)`

Adresse ≪forgée≫ détectée, arrêt de l'exécution

Exemple Défensif 1

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int(12)`	`Ref(138)`
* `0: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `1: ifle 4`	∅	`Int(12)`	`Ref(138)`
* `2: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `3: Istore 1`	∅	`Int(12)`	`Int(12)`
* `4: aload 1`	Bug	`Int(12)`	`Int(12)`

Adresse ≪forgée≫ détectée, arrêt de l'exécution

Exemple Défensif 1

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int(12)`	`Ref(138)`
* `0: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `1: ifle 4`	∅	`Int(12)`	`Ref(138)`
* `2: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
* `3: Istore 1`	∅	`Int(12)`	`Int(12)`
* `4: aload 1`	Bug	`Int(12)`	`Int(12)`

Adresse ≪forgée≫ détectée, arrêt de l'exécution

Exemple Défensif 1

`B` < `A` `A.F` est de type `A` et `B.F` aussi par héritage.

		stack	r_0	r_1(adresse)
		∅	`Int(12)`	`Ref(138)`
*	`0: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
*	`1: ifle 4`	∅	`Int(12)`	`Ref(138)`
*	`2: Iload 0`	`Int(12)`	`Int(12)`	`Ref(138)`
*	`3: Istore 1`	∅	`Int(12)`	`Int(12)`
*	`4: aload 1`	Bug	`Int(12)`	`Int(12)`

Adresse «forgée» détectée, arrêt de l'exécution

Exemple Défensif 2

`B` < `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int(-12)`	`Ref(138)`
* `0: Iload 0`	`Int(-12)`	`Int(-12)`	`Ref(138)`
* `1: ifle 4`	∅	`Int(-12)`	`Ref(138)`
`2: Iload 0`			
`3: Istore 1`			
*`4: aload 1`	`Ref(138)`	`Int(-12)`	`Ref(138)`

Pas d'adresse «forgée» détectée

Exemple Défensif 2

`B` < `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int(-12)`	`Ref(138)`
* `0: Iload 0`	`Int(-12)`	`Int(-12)`	`Ref(138)`
* `1: ifle 4`	∅	`Int(-12)`	`Ref(138)`
`2: Iload 0`			
`3: Istore 1`			
* `4: aload 1`	`Ref(138)`	`Int(-12)`	`Ref(138)`

Pas d'adresse «forgée» détectée

Exemple Défensif 2

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

		stack	r_0	r_1(adresse)
		$\emptyset$	`Int(-12)`	`Ref(138)`
*	`0: Iload 0`	`Int(-12)`	`Int(-12)`	`Ref(138)`
*	`1: ifle 4`	$\emptyset$	`Int(-12)`	`Ref(138)`
	`2: Iload 0`			
	`3: Istore 1`			
*	`4: aload 1`	`Ref(138)`	`Int(-12)`	`Ref(138)`

Pas d'adresse «forgée» détectée

Exemple Défensif 2

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

		stack	r_0	r_1(adresse)
		$\emptyset$	`Int(-12)`	`Ref(138)`
*	`0: Iload 0`	`Int(-12)`	`Int(-12)`	`Ref(138)`
*	`1: ifle 4`	$\emptyset$	`Int(-12)`	`Ref(138)`
	`2: Iload 0`			
	`3: Istore 1`			
*	`4: aload 1`	`Ref(138)`	`Int(-12)`	`Ref(138)`

Pas d'adresse «forgée» détectée

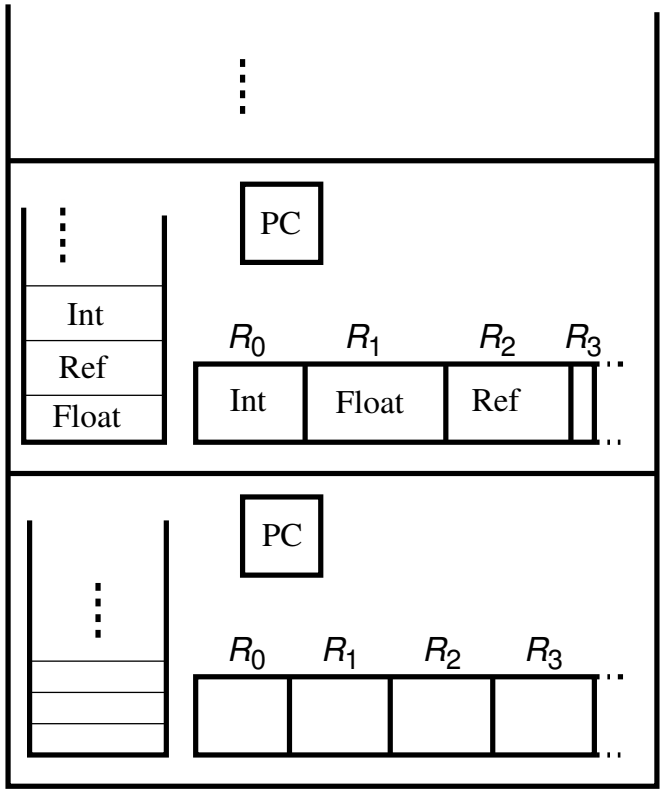
Vérifications

- Vérifier toutes les exécutions: impossible
- Exécutions ≪abstraites≫ représentant des ensembles d'exécutions concrètes
- Les valeurs disparaissent, les types restent $\Rightarrow$ Ensembles d'états

Machine *abstraite*

Machine abstraite

- Fait partie du vérificateur
- Manipule uniquement les *types*
- Informations sur les valeurs, sans les valeur (abstraction)
- Vérification des propriétés voulues pendant l'exécution abstraite



Exemple Abstrait

`B` < `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int`	`Ref`
* `0: Iload 0`	`Int`	`Int`	`Ref`
* `1: ifle 4`	∅	`Int`	`Ref`
* `2: Iload 0`	`Int`	`Int`	`Ref`
* `3: Istore 1`	∅	`Int`	`Ref`
	∅	`Int`	`Int`
* `4: aload 1`	`Int`	`Int`	`Int`
	Bug	`Int`	`Ref`

Adresse «forgée» détectée, arrêt de l'exécution

Exemple Abstrait

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	`Int`	`Ref`
* `0: Iload 0`	`Int`	`Int`	`Ref`
* `1: ifle 4`	$\emptyset$	`Int`	`Ref`
* `2: Iload 0`	`Int`	`Int`	`Ref`
* `3: Istore 1`	$\emptyset$	`Int`	`Ref`
	$\emptyset$	`Int`	`Int`
* `4: aload 1`	`Int`	`Int`	`Int`
	`Bug`	`Int`	`Ref`

Adresse «forgée» détectée, arrêt de l'exécution

Exemple Abstrait

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	`Int`	`Ref`
* `0: Iload 0`	`Int`	`Int`	`Ref`
* `1: ifle 4`	$\emptyset$	`Int`	`Ref`
*`2: Iload 0`	`Int`	`Int`	`Ref`
*`3: Istore 1`	$\emptyset$	`Int`	`Ref`
	$\emptyset$	`Int`	`Int`
*`4: aload 1`	`Int`	`Int`	`Int`
	Bug	`Int`	`Ref`

Adresse «forgée» détectée, arrêt de l'exécution

Exemple Abstrait

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	`Int`	`Ref`
* `0: Iload 0`	`Int`	`Int`	`Ref`
* `1: ifle 4`	$\emptyset$	`Int`	`Ref`
*`2: Iload 0`	`Int`	`Int`	`Ref`
*`3: Istore 1`	$\emptyset$	`Int`	`Ref`
	$\emptyset$	`Int`	`Int`
*`4: aload 1`	`Int`	`Int`	`Int`
	Bug	`Int`	`Ref`

Adresse «forgée» détectée, arrêt de l'exécution

Exemple Abstrait

`B` < `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	∅	`Int`	`Ref`
* `0: Iload 0`	`Int`	`Int`	`Ref`
* `1: ifle 4`	∅	`Int`	`Ref`
*`2: Iload 0`	`Int`	`Int`	`Ref`
*`3: Istore 1`	∅	`Int`	`Ref`
	∅	`Int`	`Int`
*`4: aload 1`	`Int`	`Int`	`Int`
	Bug	`Int`	`Ref`

Adresse «forgée» détectée, arrêt de l'exécution

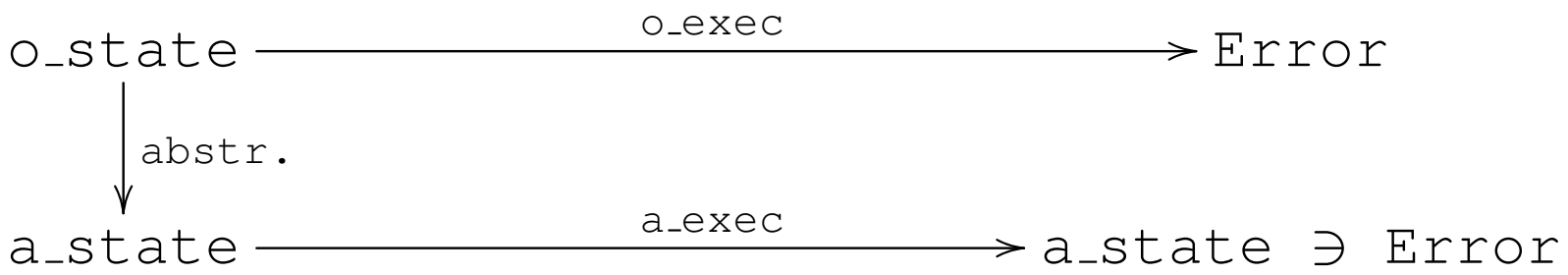
Exemple Abstrait

`B` $<$ `A` `A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1(adresse)
	$\emptyset$	`Int`	`Ref`
* `0: Iload 0`	`Int`	`Int`	`Ref`
* `1: ifle 4`	$\emptyset$	`Int`	`Ref`
*`2: Iload 0`	`Int`	`Int`	`Ref`
*`3: Istore 1`	$\emptyset$	`Int`	`Ref`
	$\emptyset$	`Int`	`Int`
*`4: aload 1`	Int	`Int`	`Int`
	Bug	`Int`	`Ref`

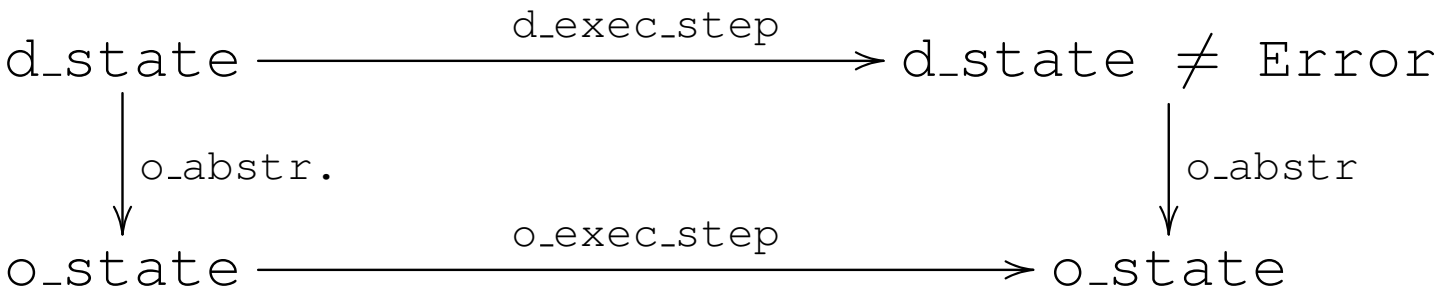
Adresse «forgée» détectée, arrêt de l'exécution

Correction d'une méthode



→ Assistant de preuve (Coq, Isabelle)

Correction pas à pas via la machine défensive



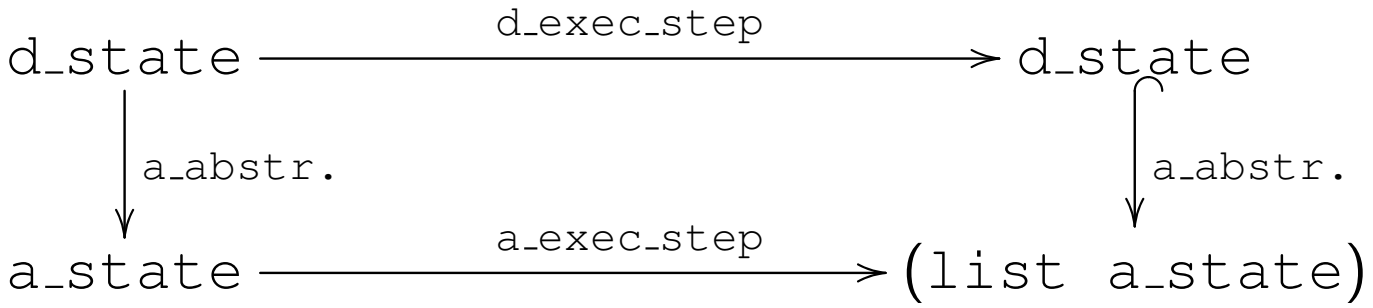


Table des matières

Principes

- Faire une analyse statique
- Construire l'ensemble d'états *entrants* possibles pour chaque point du programme
- Approximer (par au-dessus) ces ensembles
- Vérification = vérifier qu'il n'y a pas d'erreur d'exécution pour les éléments de ces ensembles

Approximation

- Types = Approximation d'ensemble de valeurs possibles pour les cases de la pile et les registres
- Problème: Deux chemins peuvent mener à des types différents pour une même cases
  - Si incompatibles alors la case est inutilisable dans la suite
  - Si il existe un *sur-type* commun (classes), alors la case peut être utilisée avec ce sur-type
  - Sur-type = Approximation de l'union de deux types

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

stack	r_0	r_1	r_2
$\emptyset$	Class B	int	int

* 0: Rload 0

* 1: Rstore 1

* 2: Rload 1

* 3: Getfield F A

* 4: Goto 1

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class B	Class B	int	int
* 1: Rstore 1				
* 2: Rload 1				
* 3: Getfield F A				
* 4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class B	Class B	int	int
* 1: Rstore 1	$\emptyset$	Class B	Class B	int
*2: Rload 1				
* 3: Getfield F A				
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack		r_0	r_1	r_2
		$\emptyset$	Class B	int	int
* 0: Rload 0	Class B		Class B	int	int
* 1: Rstore 1		$\emptyset$	Class B	Class B	int
*2: Rload 1	Class B		Class B	Class B	int
* 3: Getfield F A					
*4: Goto 1					

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack		r_0	r_1	r_2
		$\emptyset$	Class B	int	int
* 0: Rload 0	Class B		Class B	int	int
* 1: Rstore 1		$\emptyset$	Class B	Class B	int
*2: Rload 1	Class B		Class B	Class B	int
* 3: Getfield F A	Class A		Class B	Class B	int
*4: Goto 1					

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class B	Class B	int	int
	Class A	Class B	Class B	int
* 1: Rstore 1	$\emptyset$	Class B	Class B	int
*2: Rload 1	Class B	Class B	Class B	int
* 3: Getfield F A	Class A	Class B	Class B	int
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack		r_0	r_1	r_2
		$\emptyset$	Class B	int	int
* 0: Rload 0	Class A		Class B	$\top$	int
* 1: Rstore 1		$\emptyset$	Class B	Class B	int
*2: Rload 1	Class B		Class B	Class B	int
* 3: Getfield F A	Class A		Class B	Class B	int
*4: Goto 1					

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack		r_0	r_1	r_2
		$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int	
* 1: Rstore 1		$\emptyset$	Class B	Class B	int
		$\emptyset$	Class B	Class A	int
*2: Rload 1	Class B	Class B	Class B	Class B	int
* 3: Getfield F A	Class A	Class B	Class B	Class B	int
*4: Goto 1					

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack		r_0	r_1	r_2
		$\emptyset$	Class B	int	int
* 0: Rload 0	Class A		Class B	$\top$	int
* 1: Rstore 1		$\emptyset$	Class B	Class A	int
*2: Rload 1	Class B		Class B	Class B	int
* 3: Getfield F A	Class A		Class B	Class B	int
*4: Goto 1					

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int
* 1: Rstore 1	$\emptyset$	Class B	Class A	int
*2: Rload 1	Class B	Class B	Class B	int
	Class A	Class B	Class A	int
* 3: Getfield F A	Class A	Class B	Class B	int
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int
* 1: Rstore 1	$\emptyset$	Class B	Class A	int
*2: Rload 1	Class A	Class B	Class A	int
* 3: Getfield F A	Class A	Class B	Class B	int
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int
* 1: Rstore 1	$\emptyset$	Class B	Class A	int
*2: Rload 1	Class A	Class B	Class A	int
* 3: Getfield F A	Class A	Class B	Class B	int
	Class A	Class B	Class A	int
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int
* 1: Rstore 1	$\emptyset$	Class B	Class A	int
*2: Rload 1	Class A	Class B	Class A	int
* 3: Getfield F A	Class A	Class B	Class A	int
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

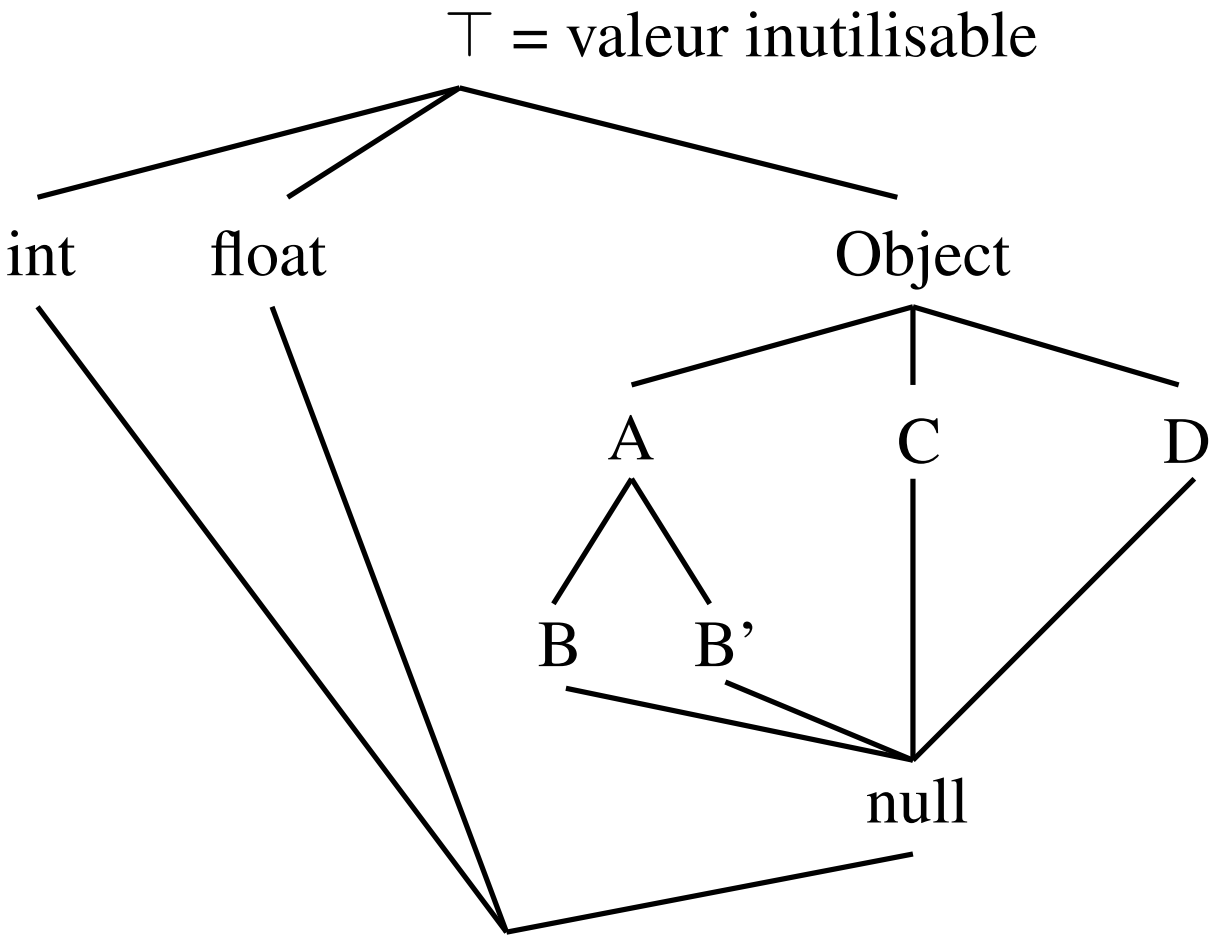
	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int
	Class A	Class B	Class B	
* 1: Rstore 1	$\emptyset$	Class B	Class A	int
*2: Rload 1	Class A	Class B	Class A	int
* 3: Getfield F A	Class A	Class B	Class A	int
*4: Goto 1				

Exemple 1

$B < A$ A.F est de type A et B.F aussi par héritage.

	stack	r_0	r_1	r_2
	$\emptyset$	Class B	int	int
* 0: Rload 0	Class A	Class B	$\top$	int
* 1: Rstore 1	$\emptyset$	Class B	Class A	int
*2: Rload 1	Class A	Class B	Class A	int
* 3: Getfield F A	Class A	Class B	Class A	int
*4: Goto 1	$\Rightarrow$ Fin			

Treillis des types



⊥ = registre jamais assigné
(ne pas confondre avec objet non initialisé)

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0			
* 1: ifle 4			
* 2: lload 0			
* 3: lstore 1			
* 4: rload 1			

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0	int	int	float
* 1: ifle 4			
*2: lload 0			
* 3: lstore 1			
*4: rload 1			

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0	int	int	float
* 1: ifle 4	$\emptyset$	int	float
*2: lload 0			
* 3: lstore 1	$\emptyset$	int	float
*4: rload 1			

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0	int	int	float
* 1: ifle 4	$\emptyset$	int	float
*2: lload 0	int	int	float
* 3: lstore 1	$\emptyset$	int	float
*4: rload 1			

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0	int	int	float
* 1: ifle 4	$\emptyset$	int	float
* 2: lload 0	int	int	float
* 3: lstore 1	$\emptyset$	int	float
	$\emptyset$	int	int
* 4: rload 1			

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0	int	int	float
* 1: ifle 4	$\emptyset$	int	float
*2: lload 0	int	int	float
* 3: lstore 1	$\emptyset$	int	$\top$
*4: rload 1			

Exemple 2

`B` $<$ `A`

`A.F` est de type `A` et `B.F` aussi par héritage.

	stack	r_0	r_1
	$\emptyset$	int	float
* 0: lload 0	int	int	float
* 1: ifle 4	$\emptyset$	int	float
*2: lload 0	int	int	float
* 3: lstore 1	$\emptyset$	int	$\top$
*4: rload 1	Bug!	int	$\top$

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0					
* 1: Rstore 1					
* 2: Rload 1					
* 3: Getfield F A					
* 4: Iload 2					
* 5: Ifle 1					
* 6: Del					
* 7: Rload 3					
* 8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class B	Class B	int	int	Class C
* 1: Rstore 1					
* 2: Rload 1					
* 3: Getfield F A					
* 4: Iload 2					
* 5: Ifle 1					
* 6: Del					
* 7: Rload 3					
* 8: Goto 1					

Exemple 3

$B < A < Object$ A.F : A, B.F : A. $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class B	Class B	int	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
*2: Rload 1					
*3: Getfield F A					
*4: Iload 2					
*5: Ifle 1					
*6: Del					
*7: Rload 3					
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class B	Class B	int	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
* 2: Rload 1	Class B	Class B	Class B	int	Class C
* 3: Getfield F A					
* 4: Iload 2					
* 5: Ifle 1					
* 6: Del					
* 7: Rload 3					
* 8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class B	Class B	int	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
* 2: Rload 1	Class B	Class B	Class B	int	Class C
* 3: Getfield F A	Class A	Class B	Class B	int	Class C
* 4: Iload 2					
* 5: Ifle 1					
* 6: Del					
* 7: Rload 3					
* 8: Goto 1					

Exemple 3

$B < A < Object$ A.F : A, B.F : A. $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class B	Class B	int	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
* 2: Rload 1	Class B	Class B	Class B	int	Class C
* 3: Getfield F A	Class A	Class B	Class B	int	Class C
* 4: Iload 2	int , Class A	Class B	Class B	int	Class C
* 5: Ifle 1					
* 6: Del					
* 7: Rload 3					
* 8: Goto 1					

Exemple 3

$B < A < Object$ A.F : A, B.F : A. $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class B	Class B	int	int	Class C
	Class A	Class B	Class B	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
*2: Rload 1	Class B	Class B	Class B	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del					
*7: Rload 3					
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class A	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
*2: Rload 1	Class B	Class B	Class B	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3					
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class A	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
* 2: Rload 1	Class B	Class B	Class B	int	Class C
* 3: Getfield F A	Class A	Class B	Class B	int	Class C
* 4: Iload 2	int , Class A	Class B	Class B	int	Class C
* 5: Ifle 1	Class A	Class B	Class B	int	Class C
* 6: Del	$\emptyset$	Class B	Class B	int	Class C
* 7: Rload 3	Class C	Class B	Class B	int	Class C
* 8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Class A	Class B	$\top$	int	Class C
	Class C	Class B	Class B	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
*2: Rload 1	Class B	Class B	Class B	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3	Class C	Class B	Class B	int	Class C
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Object	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
* 2: Rload 1	Class B	Class B	Class B	int	Class C
* 3: Getfield F A	Class A	Class B	Class B	int	Class C
* 4: Iload 2	int , Class A	Class B	Class B	int	Class C
* 5: Ifle 1	Class A	Class B	Class B	int	Class C
* 6: Del	$\emptyset$	Class B	Class B	int	Class C
* 7: Rload 3	Class C	Class B	Class B	int	Class C
* 8: Goto 1					

Exemple 3

$B < A < Object$ A.F : A, B.F : A. $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Object	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Class B	int	Class C
	$\emptyset$	Class B	Object	int	Class C
*2: Rload 1	Class B	Class B	Class B	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3	Class C	Class B	Class B	int	Class C
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Object	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Object	int	Class C
*2: Rload 1	Class B	Class B	Class B	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3	Class C	Class B	Class B	int	Class C
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Object	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Object	int	Class C
*2: Rload 1	Class B	Class B	Class B	int	Class C
	Object	Class B	Object	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3	Class C	Class B	Class B	int	Class C
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Object	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Object	int	Class C
*2: Rload 1	Object	Class B	Object	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3	Class C	Class B	Class B	int	Class C
*8: Goto 1					

Exemple 3

$B < A < Object$ $A.F : A, B.F : A.$ $C < Object$

	stack	r_0	r_1	r_2	r_3
	$\emptyset$	Class B	int	int	Class C
* 0: Rload 0	Object	Class B	$\top$	int	Class C
* 1: Rstore 1	$\emptyset$	Class B	Object	int	Class C
*2: Rload 1	Object	Class B	Object	int	Class C
*3: Getfield F A	Class A	Class B	Class B	int	Class C
	Type Error!	Class B	Object	int	Class C
*4: Iload 2	int , Class A	Class B	Class B	int	Class C
*5: Ifle 1	Class A	Class B	Class B	int	Class C
*6: Del	$\emptyset$	Class B	Class B	int	Class C
*7: Rload 3	Class C	Class B	Class B	int	Class C
*8: Goto 1					

Appel de méthode

```
f(i,x);     /* int f(int x, float y) */
```

bytecode: `invokestatic "class"."methode"`

- Dépiler `i` et `x`
- pc=pc+1
- Empiler frame courante dans la framestack
- Créer nouvelle frame courante avec
  - `i` et `x` dans les *registres* R_0 et R_1
  - pile vide
  - pc = 0

Retour de méthode

```
return a;
```

- `a` est sur la pile
- Dépiler `a`
- Supprimer la frame courante
- Dépiler la framestack
- Empiler `a`

Appel de méthode abstrait

```
f(i,x);
```

- Chaque méthode vérifiée séparément
- En supposant que les autres sont correctes
  - Dépiler les (types) arguments
  - Empiler le type du résultat
  - Pas d'empilement de frame

Pour la vérification:

- `invokestatic`: vérifier le type des arguments
- `return`: vérifier le type de retour

Les TPs - Ce que vous devez faire

Programmer:

1. 1er TP: La machine défensive (partez de la machine offensive). Fichier `dvm.ml`.
2. 2ème TP: La machine abstraite (partez de la machine défensive). Fichier `avm.ml`.
3. 3ème TP: Le *least upper bound* sur les valeurs abstraites (types), les piles (même longueur), les registres (mêmes cases remplies) et enfin les états. Fichier `lub.ml`.
4. Le vérificateur de bytecode. Fichier `bcv.ml`.

Les TPs - Ce que vous devez faire

Programmer:

1. 1er TP: La machine défensive (partez de la machine offensive). Fichier `dvm.ml`.
2. 2ème TP: La machine abstraite (partez de la machine défensive). Fichier `avm.ml`. √
3. 3ème TP: Le *least upper bound* sur les valeurs abstraites (types), les piles (même longueur), les registres (mêmes cases remplies) et enfin les états. Fichier `lub.ml`.
4. Le vérificateur de bytecode. Fichier `bcv.ml`. √

Le TP - Ce qui est fourni

- Les trois Machines virtuelles écrites en Ocaml (`vmrun main f1 f2...`)
  - Entrée: fichiers (texte) définissant les classes (voir rép. `tests`)
  - Exécute méthode principale avec les machine voulues:
    - `vmrun -d -o`
    - option `-bcv`: vérification des méthodes définies (implanter d'abord la vérification)
    - Affichage des étapes d'exécution (`-v`)
- Module `inout.ml(i)`:
  - Stocker les états entrants pour chaque valeur du pc
  - Stocker un état l'ajoute dans la liste des états "à traiter".
- Fonction de lub sur les classes (`Classes.lub jcs c1 c2` où `jcs` est l'ensemble des définitions de classes).

→ `doc/index.html`

Programmation

```
type jtype = Tint | Tfloat | Tref of class_id | Object | Top
           | Bottom | Trefnull

(** {2 Les instructions} *)
(** Le type des instructions. *)
type instr =
  | Ireturn | Iconst of int | Ineg | Iadd | Imul
  | Iload of reg_idx
  | Fload of reg_idx
  | Rload of reg_idx
  | Istore of reg_idx
  | Fstore of reg_idx
  | Rstore of reg_idx
  | Iifle of pc_idx
  | Igoto of pc_idx
  | Igetfield of fref
  | Iputfield of fref
  | Invokestatic of methref
  | Inew of class_id
```

Programmation

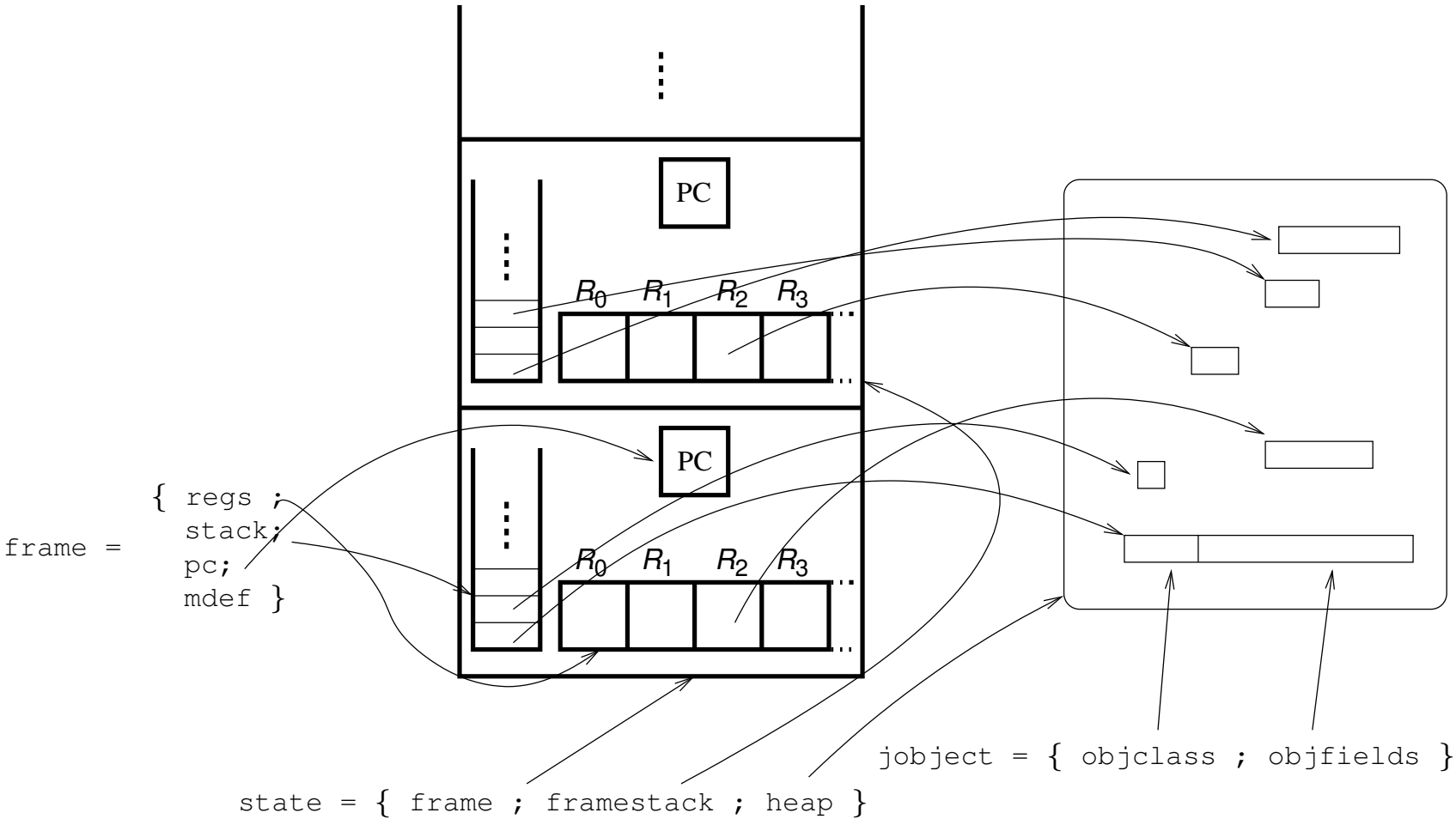
```
type 'a stack = 'a list (* La pile d'opérande *)
type 'a frame = {
  mdef  : Method.t;  (** Définition de la méthode *)
  regs  : 'a Registers.t;   (** Registres *)
  stack : 'a stack;  (** Pile d'opérande *)
  pc    : pc_idx;    (** Compteur programme *)
}
type 'a jobject = {
  objclass  : class_id;     (** Classe de l'objet *)
  objfields : 'a Fields.t;  (** Valeurs des champs *)
}
type 'a state = {
  frame      : 'a frame;      (** Frame de la méthode courante *)
  framestack : 'a framestack; (** Pile des frames des méthodes appelées *)
  heap       : 'a heap;       (** Tas *)
}
```

→ doc/index.html

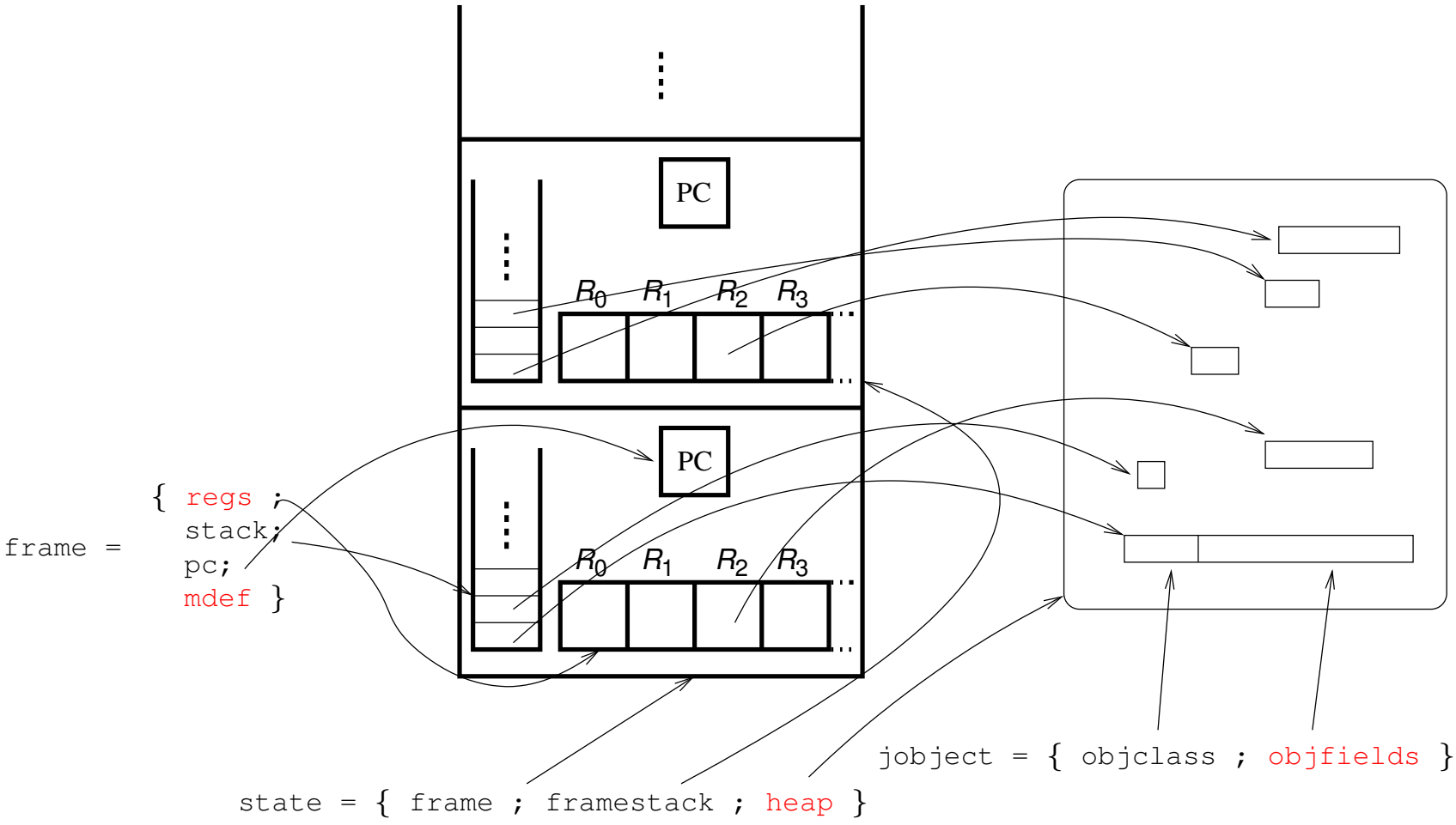
Programmation

```
type Jclass.t = {
  cname : Names.class_id;              (* nom de la classe *)
  superclass : Names.class_id option; (* nom de la classe héritée *)
  fields : Type.t Fields.t;            (* Les champs de la classes, numérotés *
  methodes : progs;                    (* l'ensemble des méthodes de la classe
  mainmethod : Names.meth_id;          (* Nom de la méthode principale *)
}
type Method.t = {
  methinclass : Names.class_id;    (* nom de la classe *)
  methname : Names.meth_id;        (* nom de la méthode *)
  methargsT : Type.t Registers.t; (* types des arguments *)
  methretT : Type.t;               (* type de retour *)
  instrs : Prog.t;                 (* les instructions de la méthode *)
}
Classes.t (* Dictionnaire (Names.class_id -> Jclass.t)*)
Fields.t  (* Dictionnaire (Names.fld_idx -> valeur) *)
Registers.t (* Dictionnaire (Names.reg_idx -> valeur) *)
```

Types de données



Types de données



Programmation

Presque toutes les structures de données sont des *Maps*.

- `<module>.add k x d` ajoute (ou remplace) `x` dans le dictionnaire `d` à l'entrée `k` (retourne le nouveau dictionnaire)
- `<module>.find k d` retourne l'élément du dictionnaire `m` à l'entrée `k`
- Exemple: `Registers.find 3 s.frame.regs`

Fonctions de recherches: `Registers.find`, `Classes.find`, `Classes.find_fld`, `Classes.find_meth`

Programmation

Pour le stockage des états en chaque point du programme, vous avez le module `Inout` (`inout.mli`).

- Type `worklist` contenant les états.
- `Inout.empty` est l'ensemble d'états vide.
- `Inout.replace_in i s wl` remplace le input state du compteur programme `i` dans `wl` par `s` et met `i` dans la liste des états à retraiter. Retourne le nouveau `worklist`.
- `Inout.pick_in wl` retourne un input state de `wl` à retraiter, ainsi que la worklist privée de l'état rendu.
- `Inout.get_in i wl` retourne l'état entrant (input state) du point de programme `i` dans la worklist `wl`.

Consulter `inout.mli` (ou mieux `doc/index.html`→Inout).

Programmation – Consulter l'état

```
(* Consulter les classes. *)
val Classes.find_fld : Names.fref -> Classes.t -> Type.t
val Classes.is_super : Classes.t -> Jclass.t -> Jclass.t -> bool
val Classes.lub : Classes.t -> Jclass.t -> Jclass.t -> Type.t
val Classes.find_meth : Names.methref -> Classes.t -> Method.t
val Jclass.find_meth : Names.methref -> Jclass.t -> Method.t

(* Consulter l'état. *)
val (XVm.)get_classfld : heap_idx -> Names.fref -> 'a state -> 'a
```

Programmation – Mise à jour d'état (record)

- Supposons que `s` est un `state`, on construit un nouveau `state` comme ceci:

```
{
  s with
  frame = {
    s.frame with
    pc=pc+1;
    stack=e::s.frame.stack
  };
  heap = Heap.add i v s.heap
}
```

- Remarquez qu'on ne donne pas tous les champs.

Programmation – Mise à jour d'état (fonction)

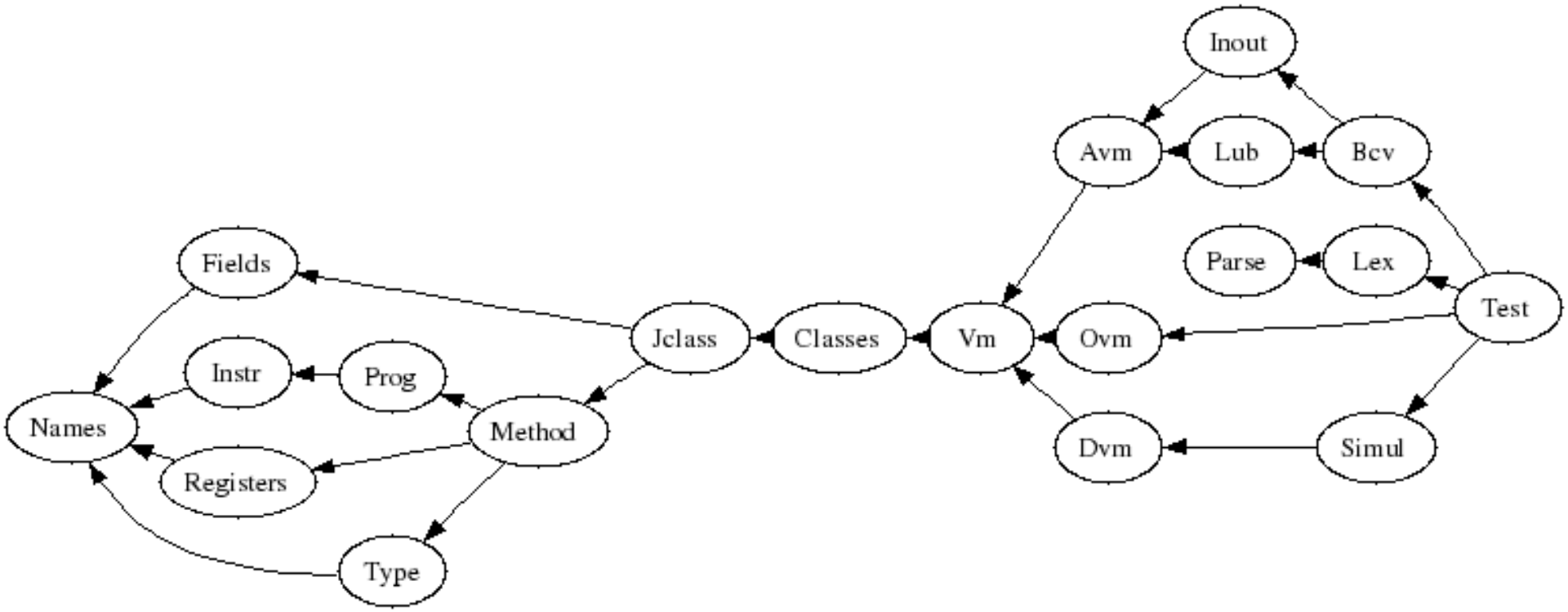
```
update_state ~pc:i ~stack:st ~regs=r ~heap=h ~fields:f s
(*  Retourne s modifié comme suit:
    - pc et la stack remplacés par i et st;
    - registres courants et tas mis à jour à l'aide de r et h ou f,
    Tous optionnels sauf s. heap et fields mutuellement exlusifs. *)

(* Exemple d'application: *)
update_state ~pc:(s.pc+3) ~regs:[(i,v1);(j,v2)] ~heap:[(k,v3);(l,v4)].
(*  Équivalent à:*)
{ s with
  frame = { s.frame with
    pc=s.pc+3;
    regs=Registers.add j v2 (Registres.add i v1 s.regs);
  };
  heap = Heap.add l v4 (Heap.add k v3 s.heap) }
```

Auto-documentation

Consulter les modules suivants:

- `Classes`
- `Inout`
- `Vm`

dépendances



Les TPs - Ce que vous devez faire

Programmer:

1. 1er TP: La machine défensive (partez de la machine offensive). Fichier `dvm.v`.
2. 2ème TP: La machine abstraite (partez de la machine défensive). Fichier `avm.v`.

Les TPs - Ce que vous devez faire

Programmer:

1. 1er TP: La machine défensive (partez de la machine offensive). Fichier `dvm.v`.
2. 2ème TP: La machine abstraite (partez de la machine défensive). Fichier `avm.v`. $\surd$

Le TP - Ce qui est fourni

- `vmtype.v` Définition des instructions
- `vmdefinition.v` Définition (générique) de l'état de la VM
- `ovm.v` Machine offensive
- `dvm.v` Squelette de machine offensive
- `avm.v` Squelette de machine abstraite

→ `doc/index.html`

Programmation

```
(** Toutes les adresses sont des entiers naturels ([nat]). *)
Definition heap_idx := nat.
Definition reg_idx := nat.
...

(** Les types, y compris les types nécessaires au vérificateur. *)
Inductive VMType:Set := Tint | Tref (id:class_id) | Object
                                    | Top | Bottom | Trefnull.

(** Profil d'une classe = type des champs (numero de champ ⟶ type). *)
Notation ClasseDef := (Dico.t VMType).

(** Ensemble de tous les profils de classes ( numero de classe ⟶ type ). *)
Notation AllClassesDef := (Dico.t ClasseDef).
```

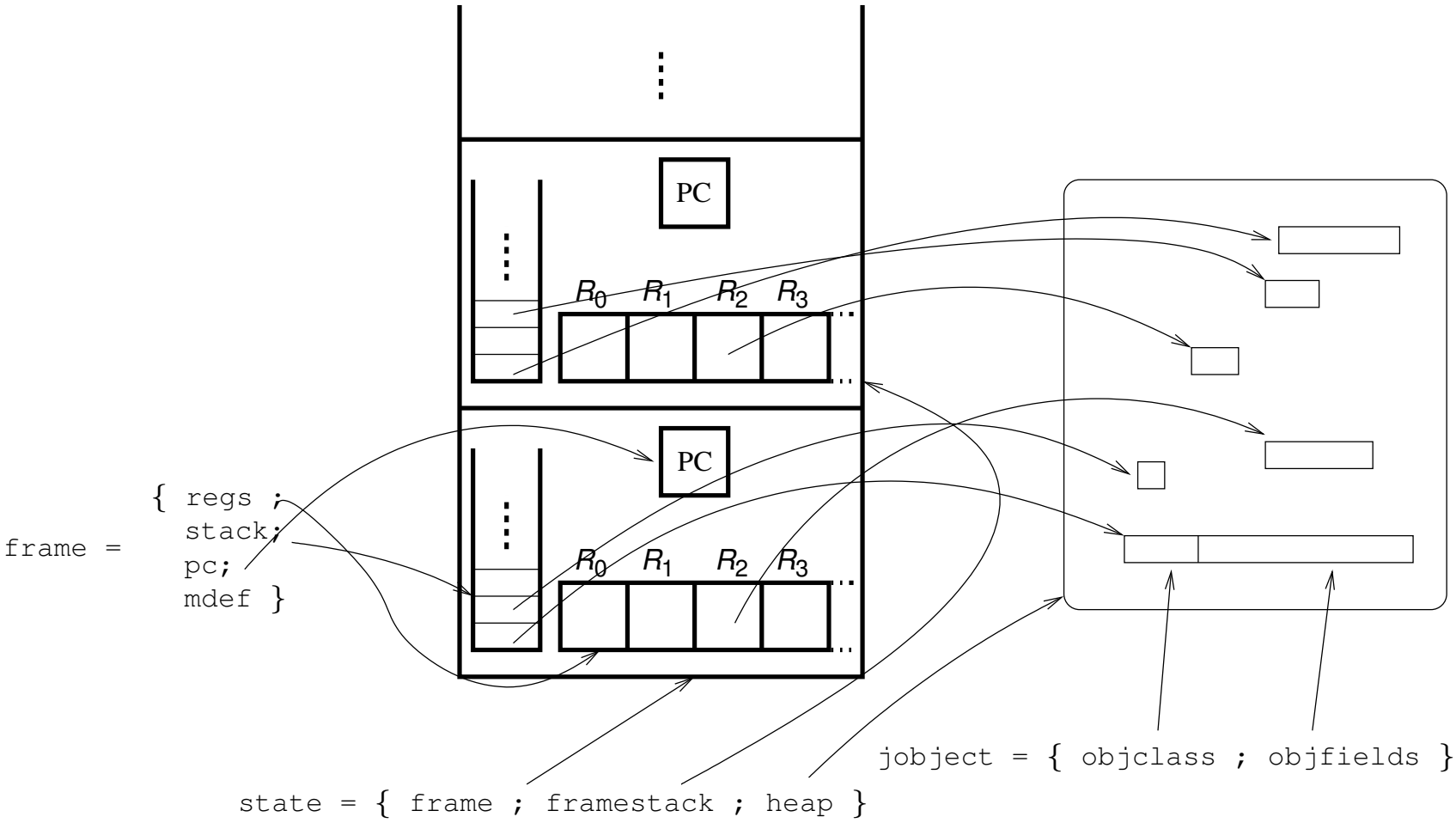
Programmation

```
(** Les instructions. Notez que les instructions sont fortement typées.
    C'est-à-dire qu'elles comportent la classe attendue sur la pile avant
    l'opération (ex: Rstore,Getfield,Putfield) et/ou la classe attendu sur
    pile après l'opération (ex: Rload,Getfield,Putfield). *)
Inductive Instr : Set :=
| Iconst (i:nat)
| Iadd
| Iload (ridx:reg_idx)
| Rload (cl:class_id) (ridx:reg_idx)
| Istore (ridx:reg_idx)
| Rstore (cl:class_id) (ridx:reg_idx)
| Iifle (pc:pc_idx)
| Goto (pc:pc_idx)
| Getfield (cl:class_id) (fldrf:fld_idx) (typ:VMType)
| Putfield (cl:class_id) (fldrf:fld_idx) (typ:VMType)
| New (clid:class_id)
| ret.


(* Méthode = dictionnaire d'instructions. *)
Notation Method := (Dico.t Instr).
```

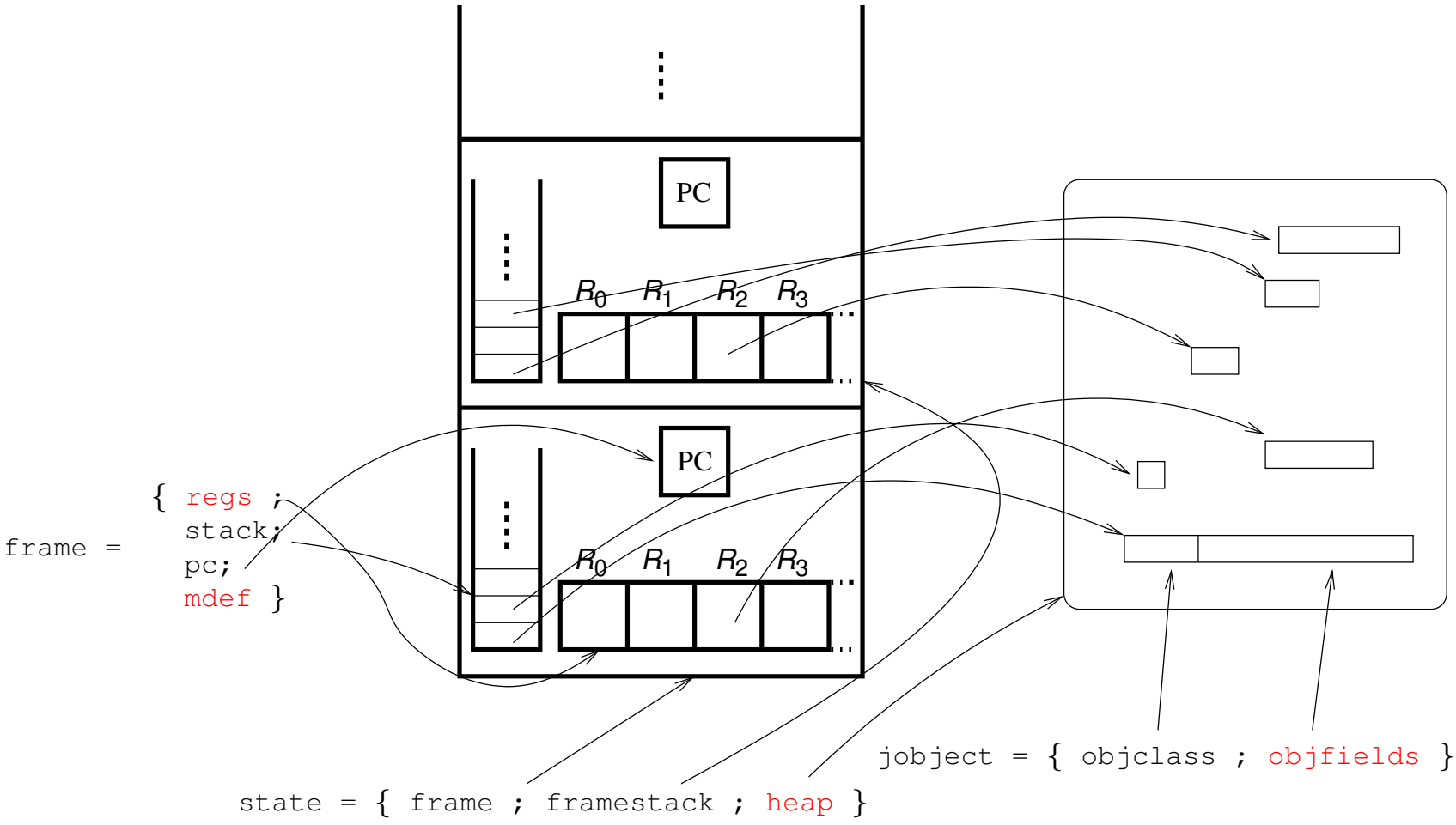
Programmation

```
Record Obj: Type := {
              objclass: class_id; (** La classe de l'objet *)
              objfields: (Dico.t Val) }.  (** les valeurs des champs *)

Notation Stack := (list Val).
Notation Registers := (Dico.t Val).
Notation Heap := (Dico.t Obj).

Record Frame : Type := Build_Frame{
                           mdef: Method;
                           regs:  Registers;
                           stack: Stack;
                           pc: pc_idx }.

Definition Framestack := list Frame.

(** L'Etat de la machine virtuelle. *)
Record State := { frame:  Frame; framestack: Framestack; heap: Heap }.
```

Types de données



Types de données



Programmation

Presque toutes les structures de données sont des *Maps* (`Dico`).

- `Dico.add k x d` ajoute (ou remplace) `x` dans le dictionnaire `d` à l'entrée `k` (retourne le nouveau dictionnaire)
- `Dico.find k d` retourne l'élément du dictionnaire `m` à l'entrée `k`
- Exemple: `Dico.find 3 s.frame.regs`