



**HAL**  
open science

## Structures de données semi-persistantes

Jean-Christophe Filliâtre

► **To cite this version:**

Jean-Christophe Filliâtre. Structures de données semi-persistantes. Doctorat. Séminaire du cours Structures de données persistantes (Xavier Leroy, chaire Sciences du logiciel), Collège de France, France. 2023. hal-04055882

**HAL Id: hal-04055882**

**<https://inria.hal.science/hal-04055882>**

Submitted on 3 Apr 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Collège de France  
Structures de données persistantes  
Séminaire

## Structures de données semi-persistantes

Jean-Christophe Filliâtre

CNRS

30 mars 2023



Sylvain Conchon

Alt-Ergo, un démonstrateur automatique écrit dans un style fonctionnel.

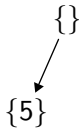
Amène notamment le problème d'une structure persistante pour les classes d'équivalence.

```
let v0 = empty ()
```

```
}
```

```
let v0 = empty ()
```

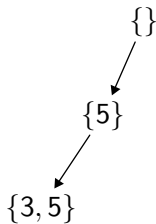
```
let v1 = add 5 v0
```



```
let v0 = empty ()
```

```
let v1 = add 5 v0
```

```
let v2 = add 3 v1
```

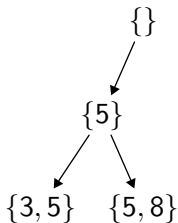


```
let v0 = empty ()
```

```
let v1 = add 5 v0
```

```
let v2 = add 3 v1
```

```
let v3 = add 8 v1
```



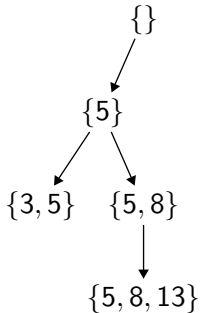
```
let v0 = empty ()
```

```
let v1 = add 5 v0
```

```
let v2 = add 3 v1
```

```
let v3 = add 8 v1
```

```
let v4 = add 13 v3
```





```
let v0 = empty ()
```

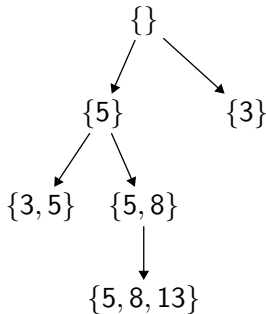
```
let v1 = add 5 v0
```

```
let v2 = add 3 v1
```

```
let v3 = add 8 v1
```

```
let v4 = add 13 v3
```

```
let v5 = add 3 v0
```



```
let v0 = empty ()
```

```
let v1 = add 5 v0
```

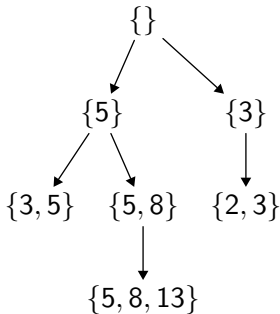
```
let v2 = add 3 v1
```

```
let v3 = add 8 v1
```

```
let v4 = add 13 v3
```

```
let v5 = add 3 v0
```

```
let v6 = add 2 v5
```



```
let v0 = create ...
```

```
let v1 = update v0 ...
```

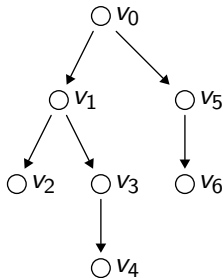
```
let v2 = update v1 ...
```

```
let v3 = update v1 ...
```

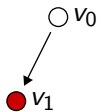
```
let v4 = update v3 ...
```

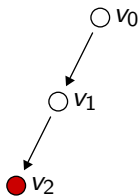
```
let v5 = update v0 ...
```

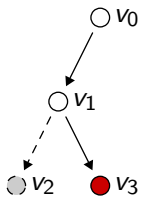
```
let v6 = update v5 ...
```

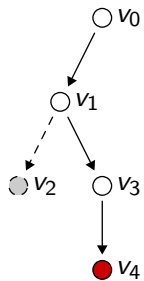


●  $v_0$

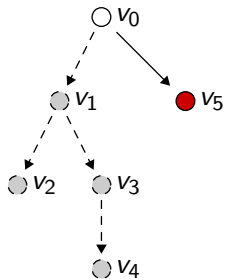


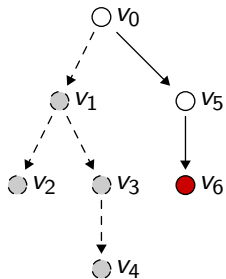


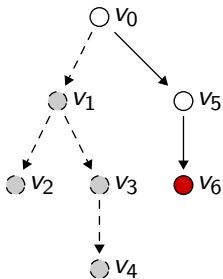












Une structure de données est dite **semi-persistante** si elle n'autorise des opérations que sur des **ancêtres** de la version courante.

## 1. Exemples de structures semi-persistantes

- tableaux
- classes d'équivalence
- listes
- graphes

## 2. Performance

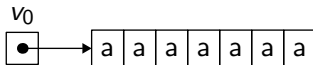
- gain en temps
- gain en mémoire

## 3. Le bon usage

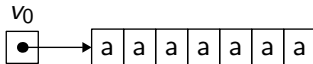
- s'assurer qu'on utilise uniquement des versions valides

```
type 'a parray =  
  'a data ref  
  
and 'a data =  
| Base of 'a array  
| Diff of int * 'a * 'a parray
```

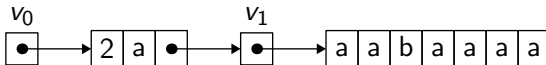
```
let v0 = create 7 a
```



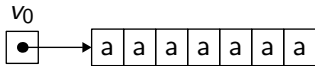
```
let v0 = create 7 a
```



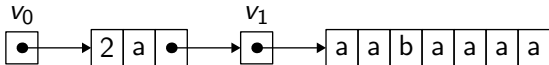
```
let v1 = set v0 2 b
```



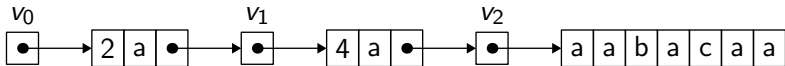
```
let v0 = create 7 a
```



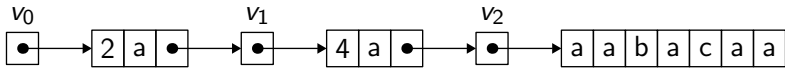
```
let v1 = set v0 2 b
```



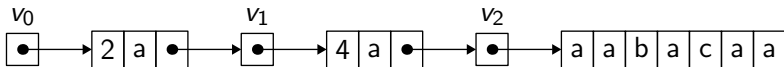
```
let v2 = set v1 4 c
```





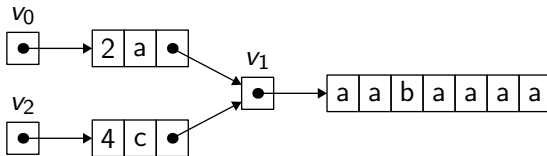


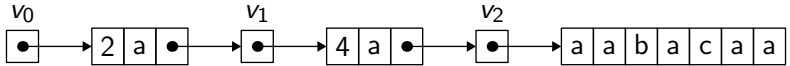
```
let v3 = set v1 5 d
```



`let v3 = set v1 5 d`

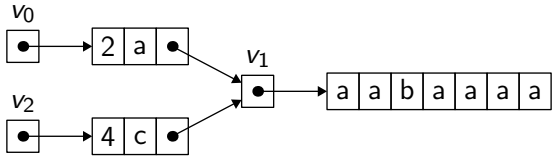
On revient à la version  $v_1$  :



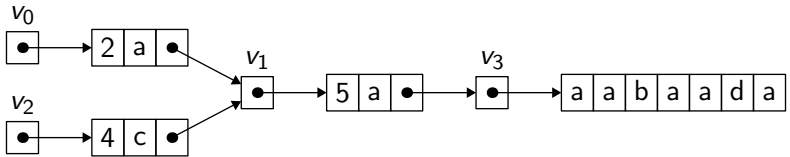


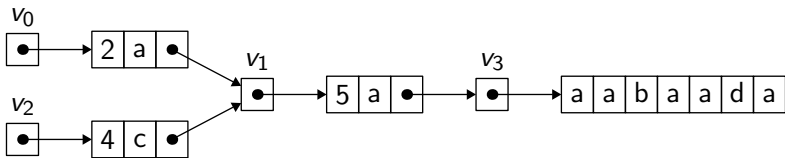
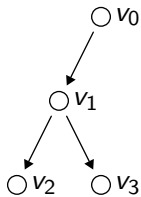
```
let v3 = set v1 5 d
```

On revient à la version  $v_1$  :



Puis on construit  $v_3$  :

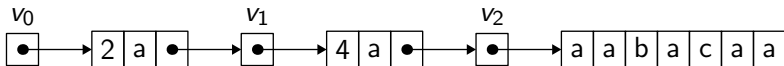
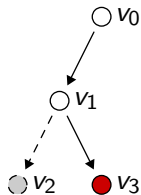




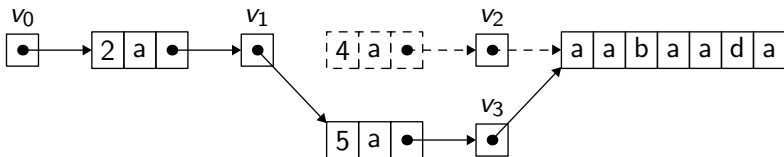
Pour rendre ces tableaux semi-persistants, il suffit de ne faire qu'une partie du travail.

- On suit la chaîne de pointeurs et on écrit les valeurs dans le tableau.
- Mais on ne cherche plus à garantir la cohérence des structures croisées en chemin.

```
let v0 = create 7 a
let v1 = set v0 2 b
let v2 = set v1 4 c
```

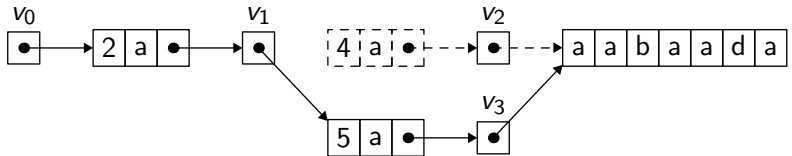
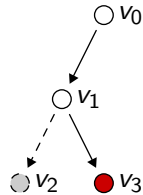


```
let v3 = set v1 5 d
```



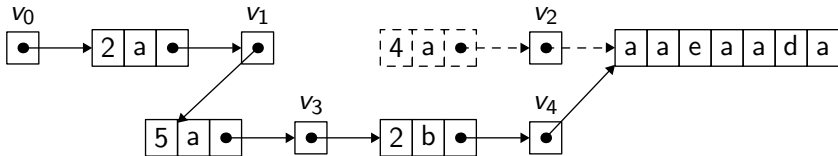
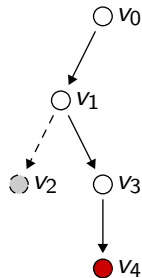
...

```
let v4 = set v3 2 e
let v5 = set v0 6 f
let v6 = set v5 3 g
```



...

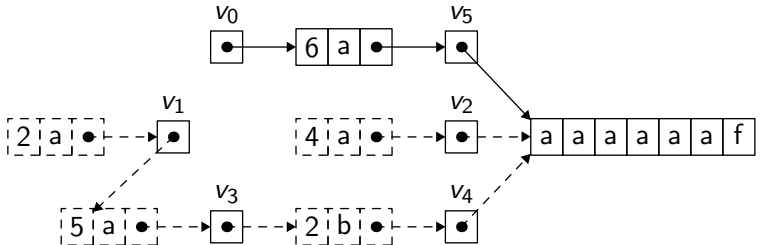
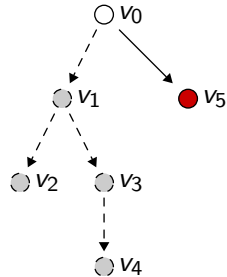
```
let v4 = set v3 2 e
let v5 = set v0 6 f
let v6 = set v5 3 g
```





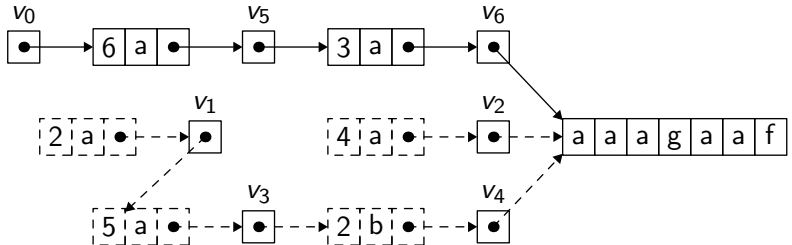
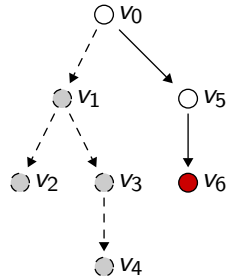
```

...
let v4 = set v3 2 e
let v5 = set v0 6 f
let v6 = set v5 3 g
    
```



```

...
let v4 = set v3 2 e
let v5 = set v0 6 f
let v6 = set v5 3 g
    
```



On a **économisé** quelques opérations (un accès dans un tableau, une allocation et une affectation).

On a **caché** les informations de retour sur trace dans la structure de données, que l'on continue à utiliser comme une structure persistante.

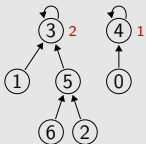
Bien entendu, on s'oblige à utiliser la structure correctement, c'est-à-dire en effectuant **uniquement des retours en arrière**.

Nous verrons plus loin comment l'assurer.

## application : classes d'équivalence

Avec deux tableaux semi-persistants, on construit une **structure semi-persistante pour les classes d'équivalence** de  $\{0, 1, \dots, N-1\}$ .

```
type spuf = {  
  link: int sparray;  
  rank: int sparray;  
}
```



Elle a l'interface d'une structure persistante :

```
type spuf  
val create: int -> spuf  
val find: spuf -> int -> spuf * int  
val union: spuf -> int -> int -> spuf
```

```
type 'a splist

val nil: unit -> 'a splist
val cons: 'a -> 'a splist -> 'a splist

val is_nil: 'a splist -> bool
val head: 'a splist -> 'a
val tail: 'a splist -> 'a splist
```

Constatation : une liste valide est un **suffixe** de la version courante.

D'où l'idée de **réutiliser** les cellules d'anciens préfixes.

```
type 'a splist = {  
  mutable head: 'a;  
           tail: 'a splist;  
  mutable prev: 'a splist;  
}
```

```
type 'a splist = {  
  mutable head: 'a;  
           tail: 'a splist;  
  mutable prev: 'a splist;  
}
```

```
let v0 = nil ()  
let v1 = cons a v0  
let v2 = cons b v1  
let v3 = cons c v1  
let v4 = cons d v3  
let v5 = cons e v0  
let v6 = cons f v5
```

v0 

⊥	⊥	⊥
---	---	---



```

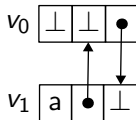
type 'a splist = {
  mutable head: 'a;
           tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```



```

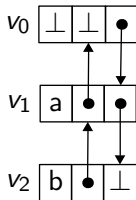
type 'a splist = {
  mutable head: 'a;
           tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```



```

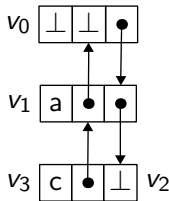
type 'a splist = {
  mutable head: 'a;
          tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```



```

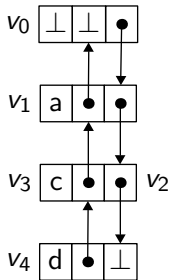
type 'a splist = {
  mutable head: 'a;
           tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```



```

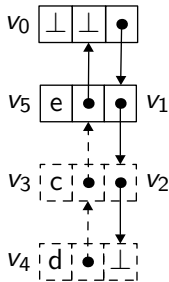
type 'a splist = {
  mutable head: 'a;
          tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```



```

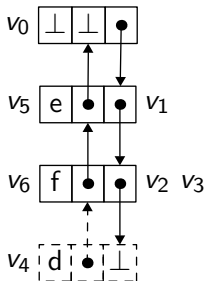
type 'a splist = {
  mutable head: 'a;
          tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```



```

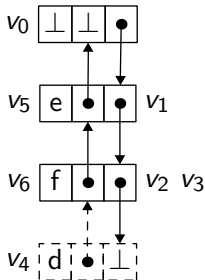
type 'a splist = {
  mutable head: 'a;
           tail: 'a splist;
  mutable prev: 'a splist;
}

```

```

let v0 = nil ()
let v1 = cons a v0
let v2 = cons b v1
let v3 = cons c v1
let v4 = cons d v3
let v5 = cons e v0
let v6 = cons f v5

```

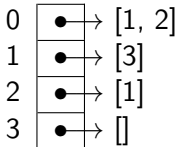
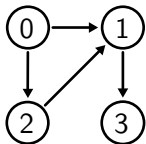


(Peut conduire à des fuites mémoire.)

## application : graphes semi-persistants

Avec des tableaux semi-persistants et des listes semi-persistantes, on peut construire une **structure semi-persistante pour des graphes** dont les sommets sont  $0, 1, \dots, N - 1$ , avec des listes d'adjacence.

```
type spgraph =  
  int splist sparray
```

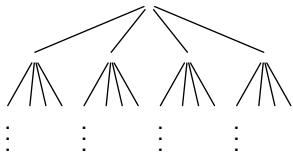




---

performance

On explore un arbre de hauteur 6 et de branchement 4.



Entre deux nœuds, on fait  $N$  opérations sur la structure de données.

Au total, on effectue donc

- $4^6 N$  opérations sur la structure ;
- $3(4^6 - 1)$  retours en arrière.

$N$	1000	5000	10 000
tableaux persistants	0,26	2,06	5,71
semi-persistants	0,13	0,90	2,27
listes persistantes	0,03	0,24	0,53
semi-persistantes	0,03	0,12	0,24
graphes persistants	0,34	3,09	8,84
semi-persistants	0,23	1,65	3,93

On mesure le nombre total de Mo alloués par le GC.

<i>N</i>	1000	5000	10 000
tableaux persistants	750	3750	7490
semi-persistants	250	1250	2500
listes persistantes	125	625	1250
semi-persistantes	0,22	1,08	2,15
graphes persistants	875	4370	8740
semi-persistants	375	1880	3750

---

le bon usage

On manipule différentes versions d'une structure de données globale d'un type `semi`, avec deux opérations :

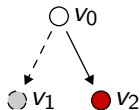
- construire une nouvelle version de la structure

```
val update: semi -> semi
```

- utiliser une version de la structure

```
val access: semi -> elt
```

Ces deux opérations exigent une version **valide**.



Correct :

```
let good (v0: semi) =  
  let v1 = update v0 in  
  let v2 = update v0 in  
  access v2
```

Incorrect :

```
let bad (v0: semi) =  
  let v1 = update v0 in  
  let v2 = update v0 in  
  access v1
```

Correct :

```
let rec backtracking (v: semi) =  
  if ... access v ... then (  
    ...  
    backtracking (update v);  
    ...  
    backtracking (update v);  
    ...  
  )
```



Il serait facile d'**invalid**er un tableau semi-persistent, en ajoutant un troisième constructeur.

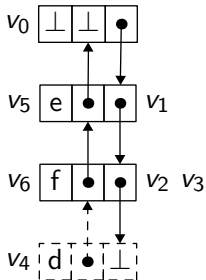
```
type 'a parray =
  'a data ref

and 'a data =
| Base of 'a array
| Diff of int * 'a * 'a parray
| Invalid
```

C'est à peine plus de travail.

C'est en revanche moins immédiat pour les listes.

Sur l'exemple suivant,



il est facile d'invalider  $v_4$ , mais moins évident d'invalider  $v_1, v_2, v_3$ .

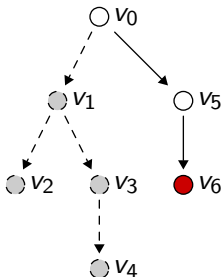
Montrons le bon usage de la structure semi-persistante en utilisant la **vérification déductive**.

1. Annoter les programmes avec un langage logique.
2. Calculer les conditions de vérification.
3. Montrer leur validité.

## Notons

- $cur$  la version courante de la structure,
- $prev(x)$  l'ancêtre immédiat de la version  $x$ ,
- $path(x, y)$  la relation «  $x$  est un ancêtre de  $y$  »,  
*i.e.*,  $path$  est la clôture réflexive transitive de  $prev^{-1}$ .

En particulier, la **validité** de la version  $x$  s'écrit  $path(x, cur)$ .



Donnons à nos fonctions un contrat formel de la forme

$$f : (x : \tau_1) \rightarrow^\varepsilon (r : \tau_2)$$

**requires**  $P$   
**ensures**  $Q$

où

- $P$  est une **précondition**
- $Q$  est une **postcondition**
- $\varepsilon$  est un **effet latent**
  - $\top$  : la version courante *cur* a (peut-être) changé
  - $\perp$  : la version courante *cur* n'a pas changé

update : (x : semi)  $\rightarrow^T$  (r : semi)  
 requires path(x, cur)  
 ensures r = cur  $\wedge$  prev(cur) = x

access : (x : semi)  $\rightarrow^T$  elt  
 requires path(x, cur)  
 ensures cur = x

```
let good (v0: semi) =  
  (* requires path(v0, cur) *)  
  let v1 = update v0 in  
  let v2 = update v0 in  
  access v2
```

$$\begin{aligned} \forall v_0. \forall cur. \text{path}(v_0, cur) &\Rightarrow \\ \text{path}(v_0, cur) \wedge & \\ \forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v_0) &\Rightarrow \\ \text{path}(v_0, cur_1) \wedge & \\ \forall v_2. \forall cur_2. (v_2 = cur_2 \wedge \text{prev}(v_2) = v_0) &\Rightarrow \\ \text{path}(v_2, cur_2) & \end{aligned}$$

```

let bad (v0: semi) =
  (* requires path(v0, cur) *)
  let v1 = update v0 in
  let v2 = update v0 in
  access v1

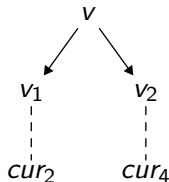
```

$$\begin{aligned}
& \forall v_0. \forall cur. \text{path}(v_0, cur) \Rightarrow \\
& \quad \text{path}(v_0, cur) \wedge \\
& \quad \forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v_0) \Rightarrow \\
& \quad \quad \text{path}(v_0, cur_1) \wedge \\
& \quad \quad \forall v_2. \forall cur_2. (v_2 = cur_2 \wedge \text{prev}(v_2) = v_0) \Rightarrow \\
& \quad \quad \quad \text{path}(v_1, cur_2)
\end{aligned}$$



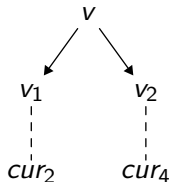
```
let rec backtracking (v: semi) =  
  (* requires path(v, cur) *)  
  (* ensures path(v, cur) *)  
  if ... access v ... then ( ...  
    backtracking (update v); ...  
    backtracking (update v); ... )
```

$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$



```
let rec backtracking (v: semi) =  
  (* requires path(v, cur) *)  
  (* ensures path(v, cur) *)  
  if ... access v ... then ( ...  
    backtracking (update v); ...  
    backtracking (update v); ... )
```

$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$   
 $\text{path}(v, cur) \wedge$



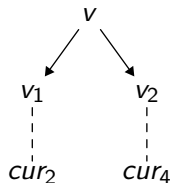
```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

$$\text{path}(v, cur) \wedge$$

$$\text{path}(v, cur) \wedge$$


```

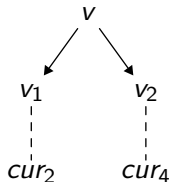
let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

$$\text{path}(v, cur) \wedge$$

$$\text{path}(v, cur) \wedge$$

$$\forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow$$


```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

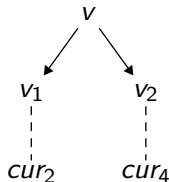
```

$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

$$\text{path}(v, cur) \wedge$$

$$\text{path}(v, cur) \wedge$$

$$\forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow$$

$$\text{path}(v_1, cur_1) \wedge$$


```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

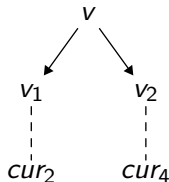
$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

$$\text{path}(v, cur) \wedge$$

$$\text{path}(v, cur) \wedge$$

$$\forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow$$

$$\text{path}(v_1, cur_1) \wedge$$

$$\forall cur_2. \text{path}(v_1, cur_2) \Rightarrow$$


```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

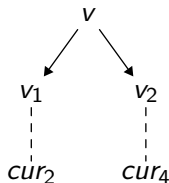
$$\text{path}(v, cur) \wedge$$

$$\text{path}(v, cur) \wedge$$

$$\forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow$$

$$\text{path}(v_1, cur_1) \wedge$$

$$\forall cur_2. \text{path}(v_1, cur_2) \Rightarrow$$

$$\text{path}(v, cur_2) \wedge$$


```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

$$\text{path}(v, cur) \wedge$$

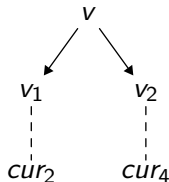
$$\text{path}(v, cur) \wedge$$

$$\forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow$$

$$\text{path}(v_1, cur_1) \wedge$$

$$\forall cur_2. \text{path}(v_1, cur_2) \Rightarrow$$

$$\text{path}(v, cur_2) \wedge$$

$$\forall v_2. \forall cur_3. (v_2 = cur_3 \wedge \text{prev}(v_2) = v) \Rightarrow$$




```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

$$\forall v. \forall cur. \text{path}(v, cur) \Rightarrow$$

$$\text{path}(v, cur) \wedge$$

$$\text{path}(v, cur) \wedge$$

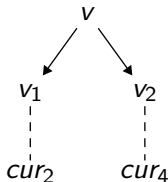
$$\forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow$$

$$\text{path}(v_1, cur_1) \wedge$$

$$\forall cur_2. \text{path}(v_1, cur_2) \Rightarrow$$

$$\text{path}(v, cur_2) \wedge$$

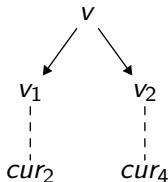
$$\forall v_2. \forall cur_3. (v_2 = cur_3 \wedge \text{prev}(v_2) = v) \Rightarrow$$

$$\text{path}(v_2, cur_3) \wedge$$


```

let rec backtracking (v: semi) =
  (* requires path(v, cur) *)
  (* ensures path(v, cur) *)
  if ... access v ... then ( ...
    backtracking (update v); ...
    backtracking (update v); ... )

```

$$\begin{aligned}
&\forall v. \forall cur. \text{path}(v, cur) \Rightarrow \\
&\quad \text{path}(v, cur) \wedge \\
&\quad \text{path}(v, cur) \wedge \\
&\quad \forall v_1. \forall cur_1. (v_1 = cur_1 \wedge \text{prev}(v_1) = v) \Rightarrow \\
&\quad \quad \text{path}(v_1, cur_1) \wedge \\
&\quad \quad \forall cur_2. \text{path}(v_1, cur_2) \Rightarrow \\
&\quad \quad \quad \text{path}(v, cur_2) \wedge \\
&\quad \quad \quad \forall v_2. \forall cur_3. (v_2 = cur_3 \wedge \text{prev}(v_2) = v) \Rightarrow \\
&\quad \quad \quad \quad \text{path}(v_2, cur_3) \wedge \\
&\quad \quad \quad \quad \forall cur_4. \text{path}(v_2, cur_4) \Rightarrow \text{path}(v, cur_4)
\end{aligned}$$


Nos conditions de vérification  $\phi$  ont une forme simple :

$$\begin{array}{ll}
 \text{termes } t & ::= x \mid \text{prev}(t) \\
 \text{atomes } a & ::= t = t \mid \text{path}(t, t) \\
 \text{cond. de vérif. } \phi & ::= a \mid \phi \wedge \phi \mid a \Rightarrow \phi \mid \forall x. \phi
 \end{array}$$

Elles doivent être valides dans la théorie  $\mathcal{T}$  qui combine l'égalité et les trois règles suivantes :

$$\frac{}{\text{path}(x, x)} \quad \frac{}{\text{path}(\text{prev}(x), x)} \quad \frac{\text{path}(x, y) \quad \text{path}(y, z)}{\text{path}(x, z)}$$

Il se trouve que déterminer

$$\mathcal{T} \models a_1 \wedge a_2 \wedge \cdots \wedge a_m \Rightarrow a$$

est **décidable**.

On veut décider la formule  $\phi \stackrel{\text{def}}{=} a_1 \wedge a_2 \wedge \dots \wedge a_m \Rightarrow a$ .

1. On construit l'ensemble  $H = \{a_1, \dots, a_m\}$ .
2. On clôt  $H$  par congruence :

$$\frac{t \in \phi}{t = t \in H} \quad \frac{t_1 = t_2 \in H}{t_2 = t_1 \in H} \quad \frac{t_1 = t_2 \in H \quad t_2 = t_3 \in H}{t_1 = t_3 \in H}$$

$$\frac{\text{prev}(t_1), \text{prev}(t_2) \in \phi \quad t_1 = t_2 \in H}{\text{prev}(t_1) = \text{prev}(t_2) \in H}$$

$$\frac{\text{path}(t_1, t_2) \in \phi \quad t_1 = t_3 \in H \quad t_2 = t_4 \in H}{\text{path}(t_3, t_4) \in H}$$

On fait cela avec une structure de **classes d'équivalence**.

On veut décider la formule  $\phi \stackrel{\text{def}}{=} a_1 \wedge a_2 \wedge \dots \wedge a_m \Rightarrow a$ .

3. Si  $a$  est de la forme  $t_1 = t_2$ ,  
on teste  $t_1 = t_2 \in H$ .
4. Si  $a$  est de la forme  $\text{path}(t_1, t_2)$ ,
  - 4.1 on construit un **graphe orienté**  $G$  dont les sommets sont les termes de  $H$  et
    - si  $\text{prev}(t) \in H$ , on a un arc  $\text{prev}(t) \rightarrow t$ ,
    - si  $\text{path}(t, t') \in H$ , on a un arc  $t \rightarrow t'$ ,
    - si  $t = t' \in H$ , on a deux arcs  $t \rightarrow t'$  et  $t' \rightarrow t$ ;
  - 4.2 on teste l'existence d'un chemin de  $t_1$  à  $t_2$  dans  $G$ .

On peut le faire **incrémentalement**, en parcourant la condition de vérification,

```
let rec decide ctx = function
  | Fatom a          -> decide_atom ctx a
  | Fand (f1, f2)    -> decide ctx f1 && decide ctx f2
  | Fimp (a, f)      -> decide (assume ctx a) f
  | Fforall f        -> decide ctx f
```

où `ctx` désigne l'état de la procédure de décision, c'est-à-dire

- une structure de classes d'équivalence ;
- un graphe.

La fonction `decide` fait un usage **semi-persistant** du contexte `ctx`.

On peut donc le réaliser ainsi :

```
type ctx = {  
  uf: spuf;  
  gr: spgraph;  
}
```

En particulier, on peut vérifier le bon usage de ce contexte par la fonction `decide`... avec la fonction `decide` elle-même !



- En pratique, on veut également permettre
  - la création dynamique de structures ;
  - l'utilisation simultanée de plusieurs ensembles disjoints de structures semi-persistantes.
- On a vérifié le bon usage d'une structure semi-persistante, mais il faut également vérifier **son implémentation**.

Une **structure semi-persistante**, c'est à la fois

- moins de possibilités qu'avec une structure persistante, mais on gagne en performances ;
- plus de possibilités qu'avec une structure éphémère, car on peut faire des retours en arrière.

On peut vérifier **mécaniquement** la bonne utilisation d'une structure semi-persistante.

- Sylvain Conchon & Jean-Christophe Filliâtre  
**Semi-Persistent Data Structures**  
ESOP 2008, LNCS vol. 4960, pp 322–336  
<https://hal.inria.fr/hal-04045849>
- Code :  
<https://github.com/backtracking/spds>
- Gabriel Scherer  
**Backtracking reference stores**  
JFLA 2023  
<https://hal.inria.fr/hal-03936704/>