Structures de données semi-persistantes

Sylvain Conchon et Jean-Christophe Filliâtre

Université Paris Sud – CNRS

LACL, 16 juin 2008

Persistance

structure de données persistante : une mise à jour renvoie un **nouvelle** structure, sans altérer son argument

- une structure purement applicative est automatiquement persistante
  $\Rightarrow$ très répandu dans la programmation fonctionnelle
- une structure de données impérative peut être rendue persistante
  [Driscoll, Sarnak, Sleator, Tarjan 89]

persistant = observationnellement immutable

(ne pas confondre avec la persistance sur le disque)

Persistance

structure de données persistante : une mise à jour renvoie un **nouvelle** structure, sans altérer son argument

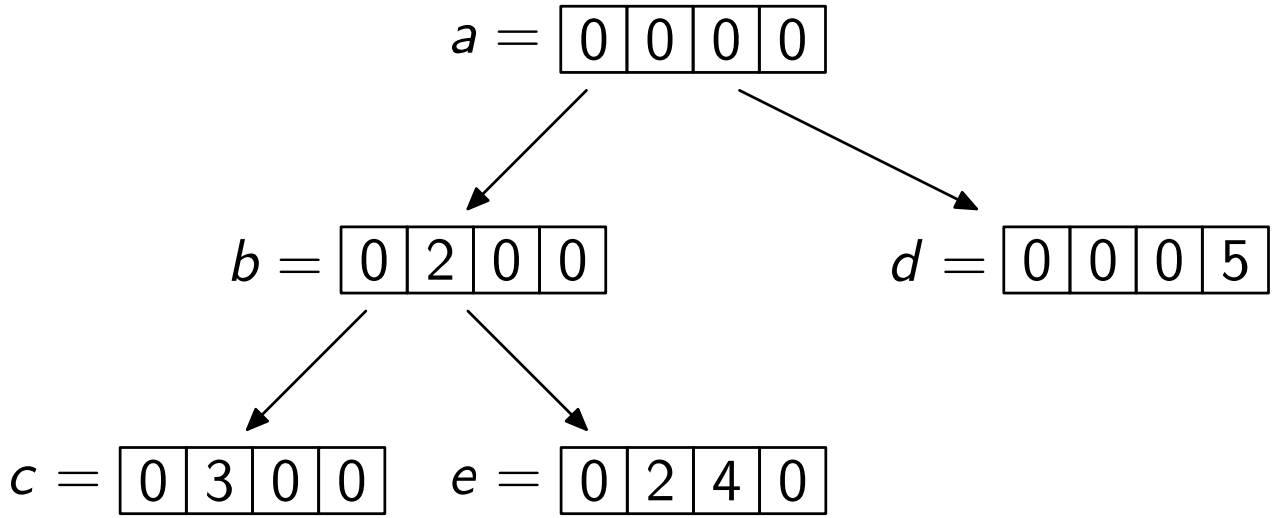
- une structure purement applicative est automatiquement persistante
  $\Rightarrow$ très répandu dans la programmation fonctionnelle
- une structure de données impérative peut être rendue persistante
  [Driscoll, Sarnak, Sleator, Tarjan 89]

$$\text{persistant} = \text{observationnellement immutable}$$

(ne pas confondre avec la persistance sur le disque)

Exemple : tableaux persistants

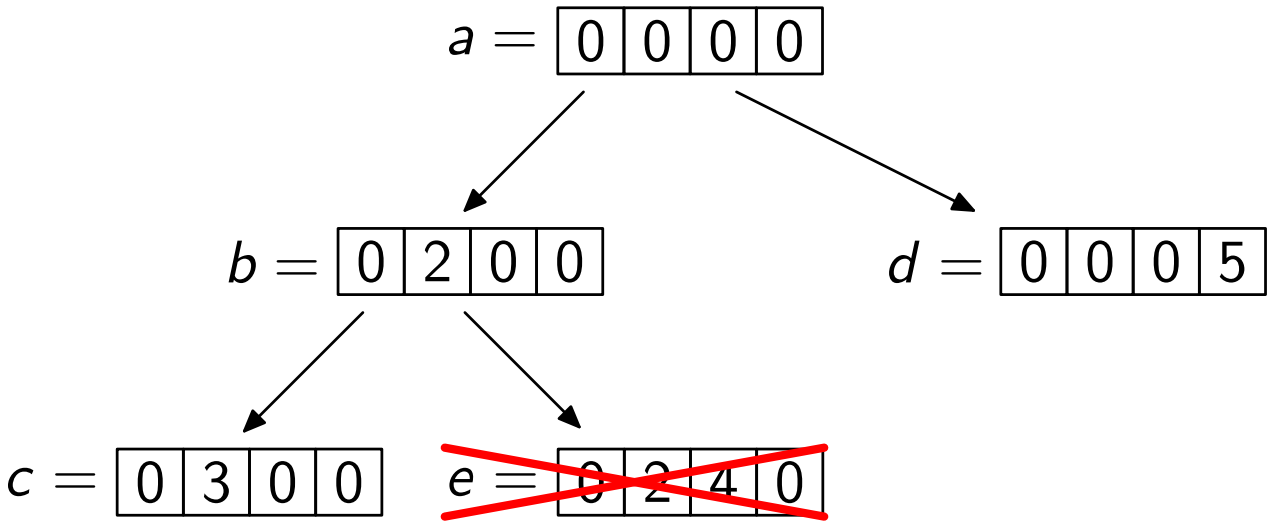
```
a := new_array(4, 0)
b := upd(a, 1, 2)
c := upd(b, 1, 3)
d := upd(a, 3, 5)
e := upd(b, 2, 4)
```



Persistance et backtracking

la persistance simplifie l'écriture des algorithmes effectuant du backtracking : pas besoin d'opération **undo** explicite, seulement la réutilisation d'une **ancienne** version de la structure

la pleine persistance n'est pas nécessaire, cependant : on revient seulement à des **ancêtres** de la version courante (arbre type parcours en profondeur)



Semi-persistance

une structure de données est dite **semi-persistante** lorsque seuls les **ancêtres de la version courante** peuvent être modifiés

(ce n'est pas la **persistance partielle** [Sleator et all], où toutes les versions peuvent être lues mais pas modifiées)

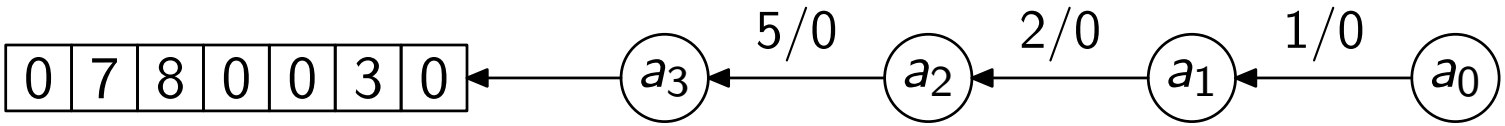
Plan

1. Exemples de structures semi-persistantes
   - tableaux, listes, tables de hachage
2. Théorie de la semi-persistance
   - vérifier qu'une structure semi-persistante est correctement utilisée

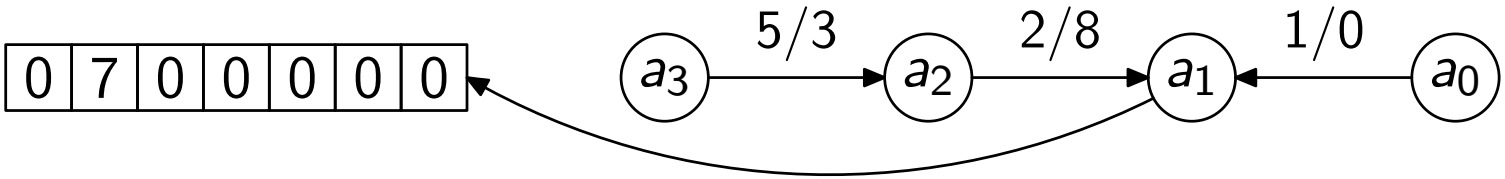
Exemple de structure semi-persistante

tableaux **persistants** à la Baker

$a_1 = \mathrm{upd}(a_0, 1, 7)$, $a_2 = \mathrm{upd}(a_1, 2, 8)$ and $a_3 = \mathrm{upd}(a_2, 5, 3)$



retour sur a_1 = retourner la liste de pointeurs, en modifiant le tableau



noter que a_3 et a_2 sont toujours valides

Tableaux persistants (code)

```
type α t = α data ref
and α data =
  | Array of α array
  | Diff of int × α × α t

let rec reroot t = match !t with
  | Array _ → ()
  | Diff (i, v, t') →
      reroot t';
      begin match !t' with
        | Array a as n →
            let v' = a.(i) in
            a.(i) ← v;
            t := n;
            t' := Diff (i, v', t)
        | Diff _ → assert false
      end
```

Tableaux persistants (code)

```
type α t = α data ref
and α data =
  | Array of α array
  | Diff of int × α × α t

let rec reroot t = match !t with
  | Array _ → ()
  | Diff (i, v, t') →
      reroot t';
      begin match !t' with
        | Array a as n →
            let v' = a.(i) in
            a.(i) ← v;
            t := n;
            t' := Diff (i, v', t)
        | Diff _ → assert false
      end
```

Tableaux persistants (code)

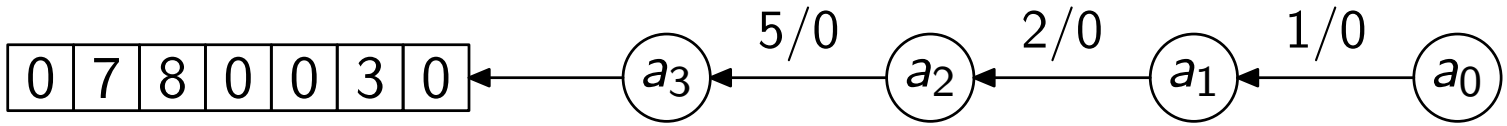
```
let get t i =
  reroot t;
  match !t with Array a → a.(i) | Diff _ → assert false

let set t i v =
  reroot t;
  match !t with
  | Array a as n →
      let old = a.(i) in
      a.(i) ← v;
      let res = ref n in
      t := Diff (i, old, res);
      res
  | Diff _ → assert false
```

Tableaux persistants (code)

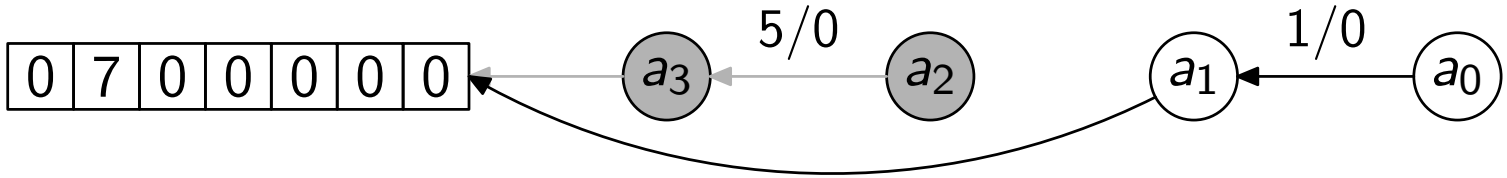
```
let get t i =
  reroot t;
  match !t with Array a → a.(i) | Diff _ → assert false

let set t i v =
  reroot t;
  match !t with
  | Array a as n →
      let old = a.(i) in
      a.(i) ← v;
      let res = ref n in
      t := Diff (i, old, res);
      res
  | Diff _ → assert false
```

Exemple de structure semi-persistante (suite)

des tableaux **semi-persistants** peuvent éviter le retournement de la liste



retour sur a_1 = on se contente de modifier le tableau



noter que a_3 et a_2 ne sont plus valides désormais

Tableaux semi-persistants (code)

```
type α t = α data ref
and α data =
  | Array of α array | Diff of int × α × α t
  | Invalid
let rec reroot t = match !t with
  | Array _ → ()
  | Diff (i, v, t') →
      reroot t' ;
      begin match !t' with
        | Array a as n →
            (* let v' = a.(i) in *)
            a.(i) ← v ;
            t := n ;
            (* t' := Diff (i, v', t) *)
            t' := Invalid
        | Diff _ → assert false
      end
```

Autre exemple : listes semi-persistantes

idée : réutiliser des **cons** déjà effectués

```
type α t = { mutable v : α ; tail : α t ; mutable succ : α t }

let cons x l =
  if is_nil l.succ then begin
    let n = { l with v=x ; tail=l } in
    if not (is_nil l) then l.succ ← n ;
    n
  end else begin
    let c = l.succ in
    c.v ← x ;
    c
  end
```

Autre exemple : listes semi-persistantes

idée : réutiliser des **cons** déjà effectués

```
type α t = { mutable v : α ; tail : α t ; mutable succ : α t }

let cons x l =
  if is_nil l.succ then begin
     let n = { l with v=x ; tail=l } in
     if not (is_nil l) then l.succ ← n ;
     n
  end else begin
    let c = l.succ in
    c.v ← x ;
    c
  end
```

Dernier exemple : tables de hachage

une simple combinaison des tableaux et des listes semi-persistantes

```
type α t = { size : int ; data : α SPL.t SPA.t }

let create n v =
  { size = n ; data = SPA.create n (SPL.nil v) }

let add h x =
  let i = x mod h.size in
  { h with data =
      SPA.set h.data i (SPL.cons x (SPA.get h.data i)) }

let mem h x =
  let i = x mod h.size in SPL.mem x (SPA.get h.data i)
```

Efficacité

tests de backtracking sur un arbre de branchement 4, de profondeur 6 et avec N opérations entre chaque nœud

N	200	1000	5000	10000
tableaux persistants	0.21	1.50	13.90	30.5
tableaux SP	0.18	1.10	7.59	17.3
listes	0.18	2.38	50.20	195.0
listes SP	0.11	0.76	8.02	31.1
tables de hachage	0.24	2.15	19.30	43.1
tables de hachage SP	0.22	1.51	11.20	28.2

Théorie de la semi-persistance

l'usage de la semi-persistance laisse l'algorithme inchangé, exactement comme s'il manipulait une structure persistante

une exigence : vérifier que l'algorithme utilise correctement la structure semi-persistante *i.e.* qu'il ne revient que sur des ancêtres de la version courante

notre solution :

- les programmes sont **annotatés** avec des pré- et post-conditions dans une **logique décidable**
- des obligations de preuve sont extraites de manière standard (WP)
- elles sont ensuite vérifiées automatiquement par une **procédure de décision**

Théorie de la semi-persistance

l'usage de la semi-persistance laisse l'algorithme inchangé, exactement comme s'il manipulait une structure persistante

une exigence : vérifier que l'algorithme utilise correctement la structure semi-persistante *i.e.* qu'il ne revient que sur des ancêtres de la version courante

notre solution :

- les programmes sont **annotatés** avec des pré- et post-conditions dans une **logique décidable**
- des obligations de preuve sont extraites de manière standard (WP)
- elles sont ensuite vérifiées automatiquement par une **procédure de décision**

Un langage simple

syntaxe

$$e \quad ::= \quad x \mid c \mid p \mid f\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$$

un programme est une ensemble de fonctions récursives

$$\texttt{fun}\ f\ (x : \iota) = \{\phi\}\ e\ \{\psi\}$$

une seule structure semi-persistante de type `semi`, avec 3 opérations :

- `backtrack` : revient sur une ancienne version, qui devient la plus récente
- `branch` : construit une nouvelle structure, successeur de la version la plus récente
- `acc` : accède en lecture une version valide quelconque

Sémantique opérationnelle

réduction petits-pas $e, S \longrightarrow e', S'$
où S, S' sont des piles de pointeurs $p_1 p_2 \cdots p_m$

les règles spécifiques sont

$$\texttt{backtrack}\ p_n\ ,\ p_1 \cdots p_n p_{n+1} \cdots p_m \longrightarrow p_n, p_1 \cdots p_n$$

$$\texttt{branch}\ p_n\ ,\ p_1 \cdots p_n \longrightarrow p, p_1 \cdots p_n p \qquad p \text{ fresh}$$

$$\texttt{acc}\ p_n\ ,\ p_1 \cdots p_n p_{n+1} \cdots p_m \longrightarrow \mathcal{A}(p_n), p_1 \cdots p_n p_{n+1} \cdots p_m$$

Typage

système de types simple avec effets ; les fonctions ont des types de la forme

$$\tau ::= (x : \iota) \rightarrow^{\epsilon} \{\phi\}\, \iota\, \{\psi\}$$

où $\epsilon \in \{\top, \bot\}$ est un booléen indiquant une modification de la structure semi-persistante et ϕ/ψ les pré/post

les effets sont utilisés dans le calcul de plus faibles pré-conditions

Une logique pour la semi-persistance

$$
\begin{array}{rrcl}
\text{termes} & t & ::= & x \mid p \mid \mathtt{prev}(t) \\
\text{atomes} & a & ::= & t = t \mid \mathtt{path}(t, t) \\
\text{post-conditions} & \psi & ::= & a \mid \psi \wedge \psi \\
\text{pré-conditions} & \phi & ::= & a \mid \phi \wedge \phi \mid \psi \Rightarrow \phi \mid \forall x.\, \phi
\end{array}
$$

$\mathtt{prev}(x)$ est l'ancêtre immédiat de x
on a $\mathtt{path}(x, y)$ si et seulement si x est un ancêtre de y

théorie $\mathcal{T}$ = combinaison de l'égalité et des axiomes suivants

$$
\begin{array}{ll}
(A_1) & \forall x.\, \mathtt{path}(x, x) \\
(A_2) & \forall xy.\, \mathtt{path}(x, \mathtt{prev}(y)) \Rightarrow \mathtt{path}(x, y) \\
(A_3) & \forall xyz.\, \mathtt{path}(x, y) \wedge \mathtt{path}(y, z) \Rightarrow \mathtt{path}(x, z)
\end{array}
$$

Une logique pour la semi-persistance

$$
\begin{array}{rrcl}
\text{termes} & t & ::= & x \mid p \mid \mathtt{prev}(t) \\
\text{atomes} & a & ::= & t = t \mid \mathtt{path}(t, t) \\
\text{post-conditions} & \psi & ::= & a \mid \psi \wedge \psi \\
\text{pré-conditions} & \phi & ::= & a \mid \phi \wedge \phi \mid \psi \Rightarrow \phi \mid \forall x.\, \phi
\end{array}
$$

$\mathtt{prev}(x)$ est l'ancêtre immédiat de x
on a $\mathtt{path}(x, y)$ si et seulement si x est un ancêtre de y

théorie $\mathcal{T}$ = combinaison de l'égalité et des axiomes suivants

$$
\begin{array}{ll}
(A_1) & \forall x.\, \mathtt{path}(x, x) \\
(A_2) & \forall xy.\, \mathtt{path}(x, \mathtt{prev}(y)) \Rightarrow \mathtt{path}(x, y) \\
(A_3) & \forall xyz.\, \mathtt{path}(x, y) \wedge \mathtt{path}(y, z) \Rightarrow \mathtt{path}(x, z)
\end{array}
$$

Validité des opérations

dans les annotations, la variable *cur* désigne la version la plus récente

la validité de x s'exprime donc par $\texttt{path}(x, cur)$

les opérations primitives ont les types suivants

$$\begin{aligned}
\texttt{backtrack}: &\quad (x : \texttt{semi}) \to^{\top} \{\texttt{path}(x, cur)\}\,\texttt{semi}\,\{ret = x \wedge cur = x\} \\
\texttt{branch}: &\quad (x : \texttt{semi}) \to^{\top} \{cur = x\}\,\texttt{semi}\,\{ret = cur \wedge \texttt{prev}(cur) = x\} \\
\texttt{acc}: &\quad (x : \texttt{semi}) \to^{\bot} \{\texttt{path}(x, cur)\}\,\delta\,\{\texttt{true}\}
\end{aligned}$$

Validité des opérations

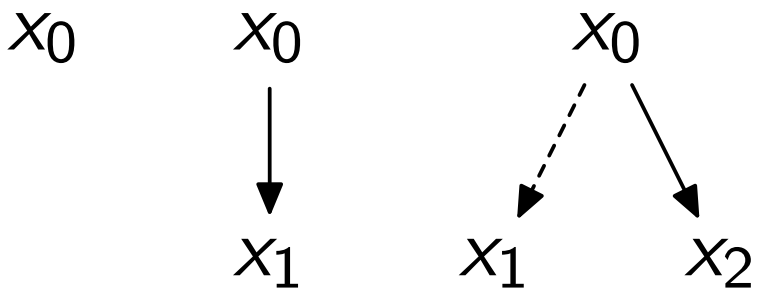
dans les annotations, la variable *cur* désigne la version la plus récente

la validité de x s'exprime donc par $\mathtt{path}(x, cur)$

les opérations primitives ont les types suivants

$$\begin{aligned}
\mathtt{backtrack}: &\quad (x : \mathtt{semi}) \rightarrow^{\top} \{\mathtt{path}(x, cur)\}\, \mathtt{semi}\, \{ret = x \wedge cur = x\} \\
\mathtt{branch}: &\quad (x : \mathtt{semi}) \rightarrow^{\top} \{cur = x\}\, \mathtt{semi}\, \{ret = cur \wedge \mathtt{prev}(cur) = x\} \\
\mathtt{acc}: &\quad (x : \mathtt{semi}) \rightarrow^{\bot} \{\mathtt{path}(x, cur)\}\, \delta\, \{\mathtt{true}\}
\end{aligned}$$

Exemple

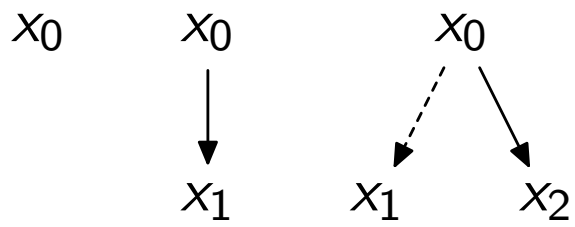


$$\texttt{fun } f_1\ (x_0 : \texttt{semi}) = \{\texttt{path}(x_0, \mathit{cur})\}\ \texttt{let } x_1 = \texttt{upd } x_0\ \texttt{in}\ \texttt{let } x_2 = \texttt{upd } x_0\ \texttt{in}\ \texttt{upd } x_2$$

$$\texttt{fun } f_2\ (x_0 : \texttt{semi}) = \{\texttt{path}(x_0, \mathit{cur})\}\ \texttt{let } x_1 = \texttt{upd } x_0\ \texttt{in}\ \texttt{let } x_2 = \texttt{upd } x_0\ \texttt{in}\ \texttt{upd } x_1$$

où $\texttt{upd } e = \texttt{branch}\ (\texttt{backtrack } e)$

Exemple : obligations de preuve



$$\texttt{fun } f_1\ (x_0 : \texttt{semi}) =$$
$$\quad \{\texttt{path}(x_0, cur)\}$$
$$\quad \texttt{let } x_1 = \texttt{upd } x_0 \texttt{ in}$$
$$\quad \texttt{let } x_2 = \texttt{upd } x_0 \texttt{ in}$$
$$\quad \texttt{upd } x_2$$

$$\forall x_0.\, \forall cur.\, \texttt{path}(x_0, cur) \Rightarrow$$
$$\quad \texttt{path}(x_0, cur) \wedge$$
$$\quad \forall r_1.\, \forall l_1.\, (\texttt{prev}(r_1) = x_0 \wedge l_1 = r_1) \Rightarrow$$
$$\quad\quad \texttt{path}(x_0, l_1) \wedge$$
$$\quad\quad \forall r_2.\, \forall l_2.\, (\texttt{prev}(r_2) = x_0 \wedge l_2 = r_2) \Rightarrow$$
$$\quad\quad\quad \texttt{path}(r_2, l_2) \wedge$$
$$\quad\quad\quad \forall r_3.\, \forall l_3.\, (\texttt{prev}(r_3) = r_2 \wedge l_3 = r_3)$$
$$\quad\quad\quad\quad \Rightarrow \texttt{true}$$

$$\texttt{fun } f_2\ (x_0 : \texttt{semi}) =$$
$$\quad \{\texttt{path}(x_0, cur)\}$$
$$\quad \texttt{let } x_1 = \texttt{upd } x_0 \texttt{ in}$$
$$\quad \texttt{let } x_2 = \texttt{upd } x_0 \texttt{ in}$$
$$\quad \texttt{upd } x_1$$

$$\forall x_0.\, \forall cur.\, \texttt{path}(x_0, cur) \Rightarrow$$
$$\quad \texttt{path}(x_0, cur) \wedge$$
$$\quad \forall r_1.\, \forall l_1.\, (\texttt{prev}(r_1) = x_0 \wedge l_1 = r_1) \Rightarrow$$
$$\quad\quad \texttt{path}(x_0, l_1) \wedge$$
$$\quad\quad \forall r_2.\, \forall l_2.\, (\texttt{prev}(r_2) = x_0 \wedge l_2 = r_2) \Rightarrow$$
$$\quad\quad\quad \texttt{path}(r_1, l_2) \wedge$$
$$\quad\quad\quad \forall r_3.\, \forall l_3.\, (\texttt{prev}(r_3) = r_2 \wedge l_3 = r_3)$$
$$\quad\quad\quad\quad \Rightarrow \texttt{true}$$

Procédure de décision

cette logique est décidable, et une procédure de décision est proposée dans l'article

Expériences :
la procédure de décision a été implantée dans le prouveur SMT **Alt-Ergo** développé dans notre équipe

la plate-forme **Why** a été utilisée pour annoter des programmes, extraire les obligations de preuve et les vérifier avec Alt-Ergo

Conclusion

nous avons proposé

- une nouvelle notion de persistance
- une logique pour vérifier son usage correct,
  ainsi qu'une procédure de décision pour cette logique

problèmes non considérés ici

- création dynamique et utilisation de plusieurs structures
  semi-persistantes

plus généralement, on peut se demander comment

- rendre une structure semi-persistante
- vérifier qu'une structure est semi-persistante