Semi-Persistent Data Structures

Sylvain Conchon and Jean-Christophe Filliâtre

Université Paris Sud – CNRS

ESOP 2008

Persistence

persistent data structure : an update operation returns a **new** data structure, **without altering** its argument

- a purely applicative data structure is automatically persistent $\Rightarrow$ widely used in functional programming
- a mutable data structure can be made persistent [Driscoll, Sarnak, Sleator, Tarjan 89]

persistent = observationally immutable

Persistence

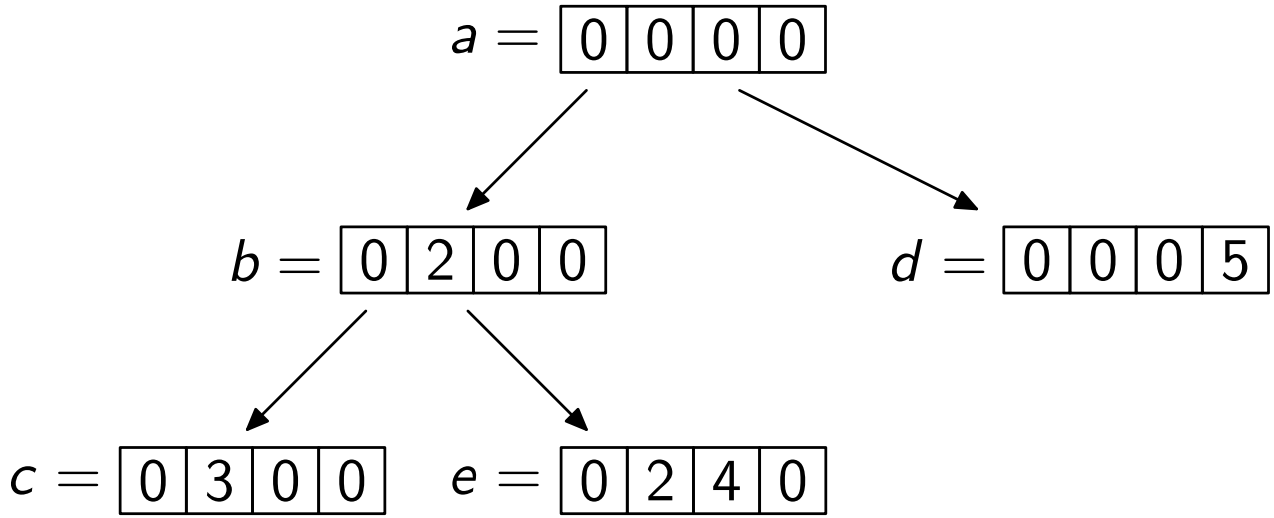
persistent data structure : an update operation returns a **new** data structure, **without altering** its argument

- a purely applicative data structure is automatically persistent $\Rightarrow$ widely used in functional programming
- a mutable data structure can be made persistent [Driscoll, Sarnak, Sleator, Tarjan 89]

persistent = observationally immutable

Example : Persistent Arrays

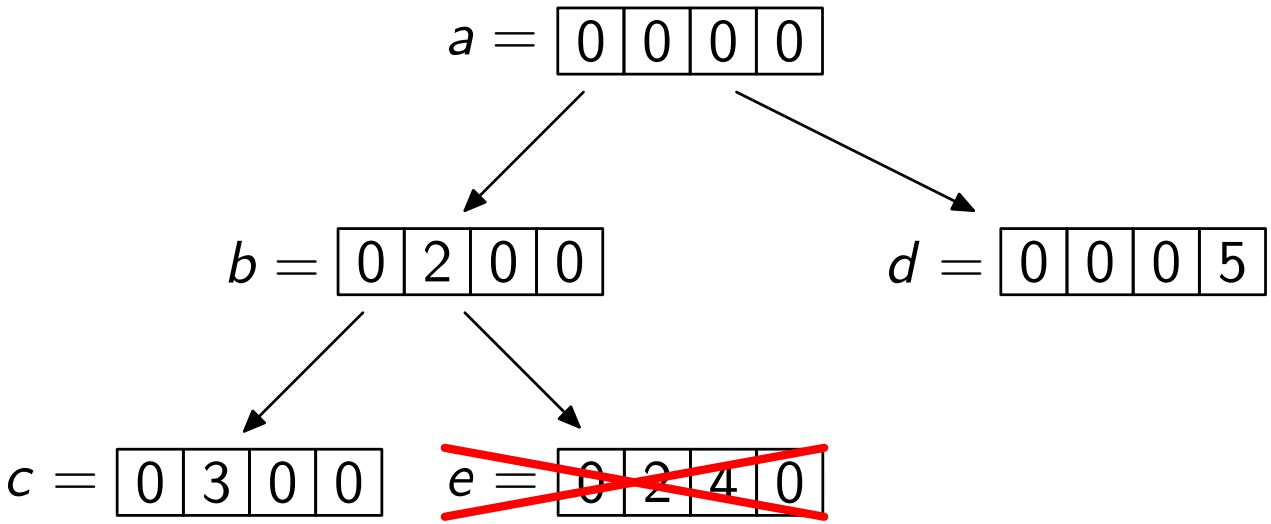
```
a := new_array(4, 0)
b := upd(a, 1, 2)
c := upd(b, 1, 3)
d := upd(a, 3, 5)
e := upd(b, 2, 4)
```



Persistence and Backtracking

persistent data structures simplify backtracking algorithms : no explicit **undo** operation but only a reuse of an **old** version

full persistence is not needed though : we only need to backtrack to **ancestors** of the current version (DFS tree)



Semi-Persistence

we call **semi-persistent** a data structure where only **ancestors of the newest version** can be updated

(this is not **partial persistence** [Sleator et al] where old versions can be accessed but not updated)

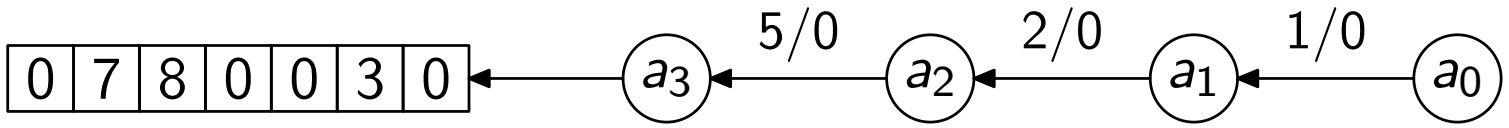
Overview

1. Examples of SP data structures
   - arrays, lists, hash tables
2. Theory of SP
   - verifying that a SP data structure is correctly used

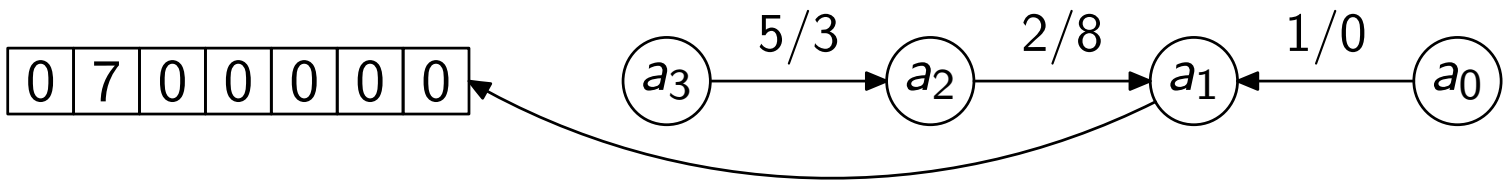
Example of SP data structure

persistent arrays *a la* Baker

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



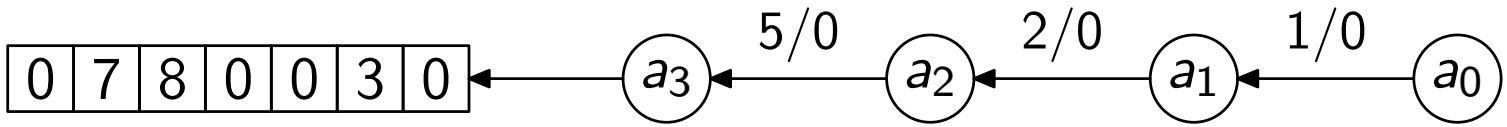
backtracking to a_1 = performing the assignments and reversing the list



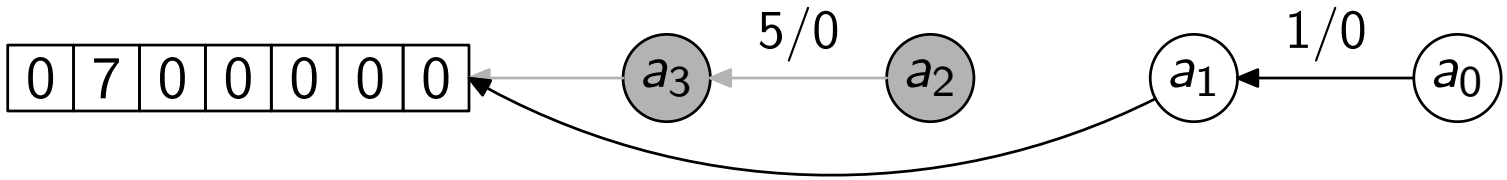
note that a_3 and a_2 are still valid

Example of SP data structure (cont'd)

semi-persistent arrays can save the list reversal



backtracking to a_1 = performing only the assignments



note that a_3 and a_2 are not valid anymore

Other Examples of SP Data Structures

- lists : reuse *cons* cells between successive conses to the same list
- hash tables : combine SP arrays and SP lists

benchmarks : backtracking of branching degree 4, depth 6 and N operations between two nodes

N	200	1000	5000	10000
persistent arrays	0.21	1.50	13.90	30.5
SP arrays	0.18	1.10	7.59	17.3
persistent lists	0.18	2.38	50.20	195.0
SP lists	0.11	0.76	8.02	31.1
persistent hash tables	0.24	2.15	19.30	43.1
SP hash tables	0.22	1.51	11.20	28.2

Other Examples of SP Data Structures

- lists : reuse *cons* cells between successive conses to the same list
- hash tables : combine SP arrays and SP lists

benchmarks : backtracking of branching degree 4, depth 6 and N operations between two nodes

N	200	1000	5000	10000
persistent arrays	0.21	1.50	13.90	30.5
SP arrays	0.18	1.10	7.59	17.3
persistent lists	0.18	2.38	50.20	195.0
SP lists	0.11	0.76	8.02	31.1
persistent hash tables	0.24	2.15	19.30	43.1
SP hash tables	0.22	1.51	11.20	28.2

Theory of Semi-Persistence

a nice feature of SP : backtracking algorithms are left unchanged (exactly as if they were manipulating persistent data structures)

requirement : we need to check that programs are using SP data structures correctly *i.e.* only backtrack to ancestors of the current version

this work addresses this issue as follows :

- programs are **annotated** with pre/post in a **decidable logic**
- verification conditions are extracted using standard techniques (WP)
- then checked using a **decision procedure**

Theory of Semi-Persistence

a nice feature of SP : backtracking algorithms are left unchanged (exactly as if they were manipulating persistent data structures)

requirement : we need to check that programs are using SP data structures correctly *i.e.* only backtrack to ancestors of the current version

this work addresses this issue as follows :

- programs are **annotated** with pre/post in a **decidable logic**
- verification conditions are extracted using standard techniques (WP)
- then checked using a **decision procedure**

A Small Language

syntax

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

a program is a set of mutually recursive functions

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

a single SP data structure of type `semi`, with three operations :

- `backtrack` : backtracks to a valid version, making it the newest
- `branch` : builds a new successor of the newest version
- `acc` : accesses any valid version

Operational Semantics

small-step reduction $e, S \to e', S'$
where S, S' are stacks of pointers $p_1 p_2 \cdots p_m$

specific reduction rules are

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

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

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

Type System

simple type system with effects ; functions are given types

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

where $\epsilon \in \{\top, \bot\}$ is a boolean indicating a modification of the SP data structure and ϕ/ψ the pre/post

effects are used in the WP calculus

Logic for Semi-Persistence

$$
\begin{array}{rrcl}
\text{terms} & t & ::= & x \mid p \mid \mathtt{prev}(t) \\
\text{atoms} & a & ::= & t = t \mid \mathtt{path}(t, t) \\
\text{postconditions} & \psi & ::= & a \mid \psi \wedge \psi \\
\text{preconditions} & \phi & ::= & a \mid \phi \wedge \phi \mid \psi \Rightarrow \phi \mid \forall x.\, \phi
\end{array}
$$

$\mathtt{prev}(x)$ is the immediate ancestor of x
$\mathtt{path}(x, y)$ holds if and only if x is an ancestor of y

theory $\mathcal{T}$ = combination of equality and the following axioms

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

Logic for Semi-Persistence

$$
\begin{array}{rrcl}
\text{terms} & t & ::= & x \mid p \mid \mathtt{prev}(t) \\
\text{atoms} & a & ::= & t = t \mid \mathtt{path}(t, t) \\
\text{postconditions} & \psi & ::= & a \mid \psi \wedge \psi \\
\text{preconditions} & \phi & ::= & a \mid \phi \wedge \phi \mid \psi \Rightarrow \phi \mid \forall x.\, \phi
\end{array}
$$

$\mathtt{prev}(x)$ is the immediate ancestor of x
$\mathtt{path}(x, y)$ holds if and only if x is an ancestor of y

theory $\mathcal{T}$ = combination of equality and the following axioms

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

Validity of Operations

within annotations, variable *cur* stands for the newest version

validity of version x is thus $\texttt{path}(x, cur)$

primitive operations have the following types :

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

Validity of Operations

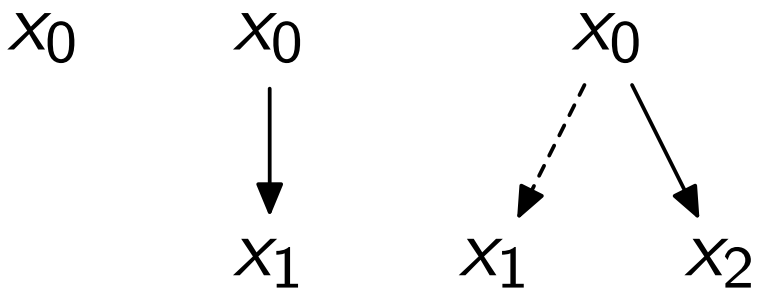
within annotations, variable *cur* stands for the newest version

validity of version x is thus $\texttt{path}(x, cur)$

primitive operations have the following types :

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

Example

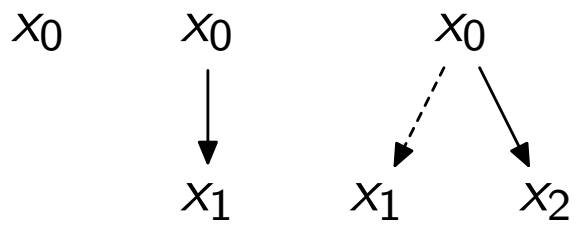


$$\texttt{fun } f_1\ (x_0 : \texttt{semi}) = \\ \quad \{\texttt{path}(x_0, \mathit{cur})\} \\ \quad \texttt{let } x_1 = \texttt{upd } x_0 \texttt{ in} \\ \quad \texttt{let } x_2 = \texttt{upd } x_0 \texttt{ in} \\ \quad \texttt{upd } x_2$$

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

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

Example : verification conditions



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

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

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

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

Implementation

our logic is decidable, and a decision procedure has been implemented in the **Alt-Ergo** theorem prover

the platform **Why** was used to annotate programs, extracting verification conditions and checking them using Alt-Ergo

Conclusion

we have proposed

- a new notion of persistence
- a logic to check the use of SP data structures, together with a decision procedure for this logic

issues not addressed

- dynamic creation and simultaneous use of several data structures

more general issues

- making a data structure SP
- verifying that a data structure is SP