# A declarative-friendly API for Web document manipulation

Benjamin Canou

## Introduction 1

A bit of background :

- Context: the Ocsigen project, typesafe multi-tiers programming in OCaml
- On the server : high level, type safe XML generation
- On the client : the DOM, low level, unsafe document modifications
- We want the same level of type safety on both parts
- The DOM makes it impossible

We propose an alternative document model

- Usable on both parts
- Compatible with high level abstractions
- Compatible with static typing

Outline of this exposé :

- Explanation of the problem
- Principle of the solution
- Formal specification of the document model
- Conclusion and future works

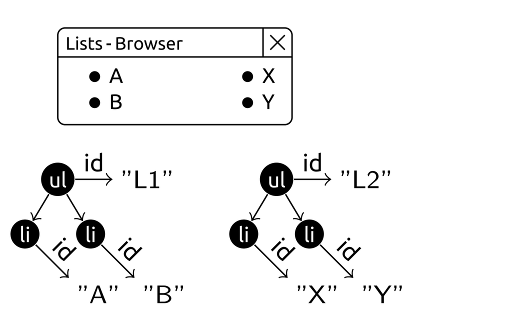
# Implicit moves in the DOM

## Implicit moves 2

- We start with from a simple, valid page

```
... <ul id="L1">
      <li id="A">A</li>
      <li id="B">B</li>
    </ul>
    <ul id="L2">
      <li id="X">X</li>
      <li id="Y">Y</li>
    </ul> ...
```

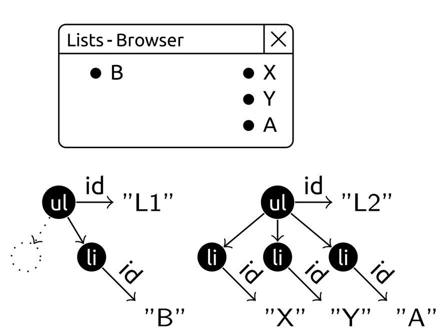
- Resulting rendering and DOM




- We execute the following JavaScript

```
var l2 = getElementById ("L2")
var a = getElementById ("A")
l2.appendChild (a) ;;
```

- Resulting in an implicit move

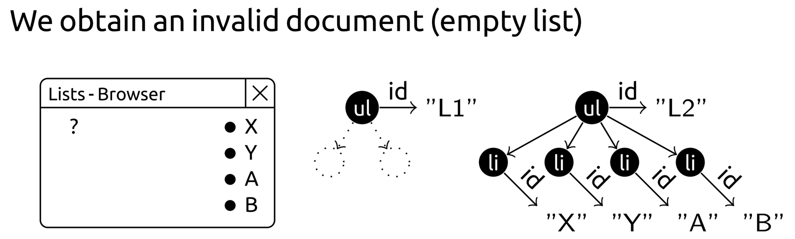



## Breaking the validity 3

- In the same page, we execute the following JavaScript instead

```
var l2 = getElementById ("L2")
var a = getElementById ("A")
var b = getElementById ("B")
  l2.appendChild (a)
  l2.appendChild (b)
```

- We obtain an invalid document (empty list)




## Breaking validity with purely constructive code 4

- We start with a HOP source code to build two lists:

```
(let ((a (<LI> "A"))    ; X
      (b (<LI> "B")))   ; Y
  (<DIV> (<UL> (a))
         (<UL> (a b)))); Z
```

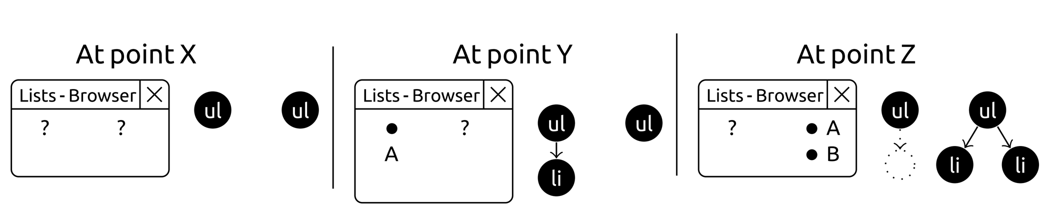
- Result on the server:

```
<DIV>
  <UL><LI>A</LI></UL>
  <UL><LI>A</LI><LI>B</LI></UL>
</DIV>
```

- Result on the client:

```
<DIV>
  <UL>            </UL>
  <UL><LI>A</LI><LI>B</LI></UL>
</DIV>
```

- Evaluation on the client:




## Summary 5

No surprise, the DOM is not a nice API for declarative programming:

- It has an unusual, difficult to predict semantics
- It breaks static typing of modification as well as construction
- It introduces differences between server and client sides
- Static detection of implicit moves is difficult

But do we, declarative programmers, really care?

- As we have seen, using the DOM directly is not an option
- Usual cheat: intermediate representation allowing sharing
- In the end, the document is always stored as a DOM
- The transition to the DOM can be delayed, but not avoided

# Presentation of our solution

## Implicit copies instead of moves 6

The idea is simple:

- Detect at run-time whenever sharing would be introduced
- Insert a copy instead of the node itself to prevent the move

The implementation not so much:

- The easy way: deep copy of the document structure only
  - As done by the DOM primitive `cloneNode(n, true)`
  - The copy looks similar but does not respond to any action
- The useful way: deep copy that includes attached objects
  - Done by some libraries but with important restrictions
  - Needs some information or convention to know which objects to copy

We need a sensible convention, here is what we propose:

- Let the programmer decide whether objects belong to a node or not
- For this, reuse a familiar notion: lexical scoping

But we want to be as generic as possible:

- We use a stratified solution: high level language + low level API
- The high level language gives a sense to the meta information
- The low level API has primitives to manipulate the meta information

In this presentation:

- We give a glimpse of our work on the high level part for the intuition
- What we present is actually the generic, low level layer: *ᶜDOM*

## Overview of the high level part 7

We introduce a delimited node definition syntax (here in an ML derivative)

- Everything allocated inside a node definition is copied along
- Everything allocated outside is shared between copies

Example: a button incrementing a counter and updating its text

- Shared counter

```
let with_shared_counter =
 let r = ref 0 in (* outside *)
  let rec self =
   node <a>
    [ node <text> content = "incr" end ]
    prop on_click = fun () ->
     r := !r + 1 ;
     replace self
       [ node <text> ()
           content = string_of_int !r
         end ]
   end
  in self ;;
```

- Local counter

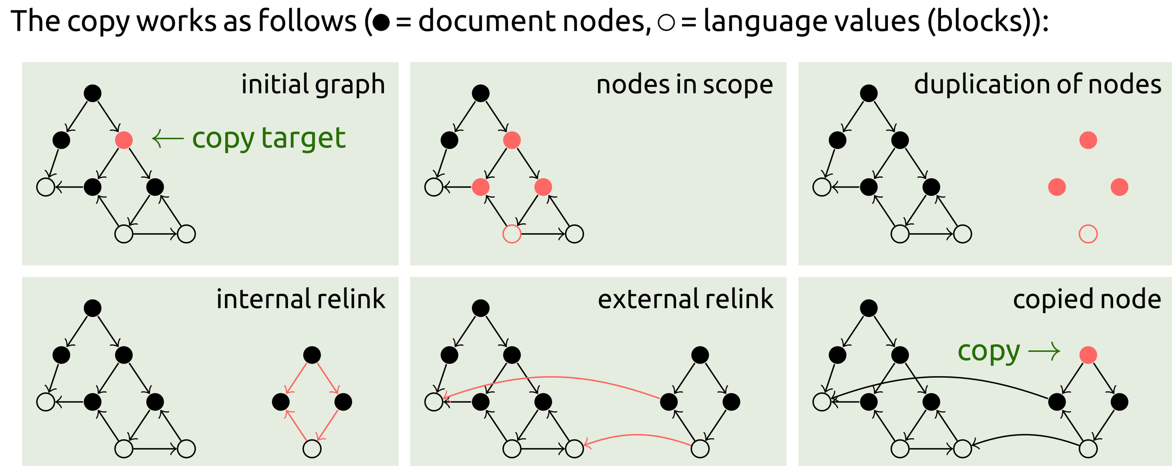
```
let with_copied_counter =
 let rec self =
  node <a>
   let r = ref 0 in (* inside *)
    [ node <text> content = "incr" end ]
    prop on_click = fun () ->
     r := !r + 1 ;
     replace self
       [ node <text> ()
           content = string_of_int !r
         end ]
  end
 in self ;;
```

## Overview of the low level part 8

Presentation of the *ᶜDOM* API:

- As low level as the DOM so it can be used as a replacement
- Can be implemented on top of the DOM
- Introduces new primitives to maintain run-time meta (scoping) information
- Performs implicit copies instead of moves

The copy works as follows (● = document nodes, ○ = language values (blocks)):




## The complete picture 9

We restore the possibilities we had on the server that where lost with the DOM:

- No more unexpected side effect can arise
- We get a much more usual semantics for declarative programmers
- We can reuse existing high level libraries for building XML
- We preserve type safety since the copy operation preserves the structure
- We can reuse existing type systems for XML

And introduce new possibilities:

- Type checking of imperative manipulations of the Web page
  - Without moves, only creations and explicit mutations have to be checked
  - Creation can be checked using existing type system for XML
  - Mutation is checked by type checking the new contents
  - The only specificity is a restriction of recursive definitions
- Explicit copy is a new tool given to the programmer
- Meta information could be used for other purposes, eg. serialization, migration

# Definition of *ᶜDOM*

## Structure of the specification 10

An API, specified as follows:

- As set of simply typed, language agnostic primitives
- Formal specification of the document state
- Operational semantics rules

of the form $\frac{(\text{RULE}) \quad \textit{conditions}}{S \vdash \texttt{prim}(a_0, \cdots, a_n) = r, S'}$

And a few properties:

- Internal consistency
- Structure preservation
  used by the high level part to ensure type preservation

The document state is specified as a tuple $(H, L, T, P, S, s)$

- Document structure: Heap, Labels, Tree and Properties
  - $H \subseteq$ *Node* $\cup$ *Block* is the domain of existing objects
  - $L \subseteq$ *Node* $\times$ *Tag* gives a tag to each node of the document
  - $T \subseteq$ *Node* $\times$ *List*(*Node*) associates to each node the list of its children
  - $P \subseteq$ *Object* $\times$ *Key* $\times$ *Value* associates objects to values through labels
- Meta (scope) information: Scopes and Stack
  - $S \subseteq$ *Node* $\times$ *Object* records for each nodes the objects under its scope
  - $s \in$ *List*(*Node*) represents the stack of currently opened scopes

With a well-formed predicate

- $L$ maps each node in $H$ to a unique tag
- $T$ is a forest (no sharing, no cycles) over $H \cap$ *Node*
- $T$ and $P$ only reference nodes present in $H$
- $P$ only references blocks present in $H$
- An object can be in the scope of only one node in $S$
- No cyclic scope chain exist in $S$.

## The API 11

- Access
  - *Int* `children` (*Node*) — *number of children on a node*
  - *Node* + *Nil* `child` (*Node*, *Int*) — *retrieve the $n^{th}$ child*
  - *Enum*(*Node*) `roots` (*Nil*) — *retrieve all nodes without parents*
  - *Enum*(*Key*) `properties` (*Object*) — *domain of properties of an object*
  - *Value* + *Nil* `get` (*Object*, *Key*) — *access to a property*
  - *Tag* `tag` (*Node*) — *retrieve the tag of a node*
- Creation
  - *Node* `create_node` (*Tag*) — *new, empty node + open its scope*
  - *Nil* `close` (*Node*) — *close the scope of a node*
  - *Object* `create_block` (*Nil*) — *new, empty block*
- Modification
  - *Nil* `reopen` (*Node*) — *repoen the scope of a node*
  - *Nil* `detach` (*Node*) — *unlink a node from its parent*
  - *Node* `copy` (*Node*) — *explicit deep copy operation*
  - *Nil* `bind` (*Node*, *Node*, *Int*) — *link a node to a parent, copy if nec.*
  - *Nil* `set` (*Object*, *Key*, *Value*) — *assign a property*
  - *Nil* `unset` (*Object*, *Key*) — *remove a property*

## Semantics 12

- Access

$$\text{(CHILD)}\ \frac{\bullet \in H \cap \textit{Node} \quad 0 \leqslant i < \textit{length}(T(\bullet))}{S \vdash \texttt{child}(\bullet, i) = \textit{nth}(T(\bullet), i), S} \qquad \text{(CHILDREN)}\ \frac{\bullet \in H \cap \textit{Node}}{S \vdash \texttt{children}(\bullet) = \textit{length}(T(\bullet)), S}$$

$$\text{(CHILD-UNBOUND)}\ \frac{\bullet \in H \cap \textit{Node} \quad \neg(0 \leqslant i < \textit{length}(T(\bullet)))}{S \vdash \texttt{child}(\bullet, i) = \textit{nil}, S}$$

$$\text{(ROOTS)}\ \frac{}{S \vdash \texttt{roots}(\textit{nil}) = \textit{enum}(\{\bullet | \textit{Anc}(\bullet) = \emptyset\}), S} \qquad \text{(TAG)}\ \frac{(\bullet, t) \in L}{S \vdash \texttt{tag}(\bullet) = t, S}$$

$$\text{(PROPERTIES)}\ \frac{\odot \in H}{S \vdash \texttt{properties}(\odot) = \textit{enum}(\{k | (\odot, k, v) \in P\}), S}$$

$$\text{(GET)}\ \frac{\exists (\odot, k, v) \in P}{S \vdash \texttt{get}(\odot, k) = v, S} \qquad \text{(GET-UNBOUND)}\ \frac{\nexists (\odot, k, v) \in P}{S \vdash \texttt{get}(\odot, k) = \textit{nil}, S}$$

- Modification

$$\text{(SET)}\ \frac{v \in H \cup \textit{Imm} \quad k \in \textit{Key} \quad \odot \in H \quad \nexists v', (\odot, k, v') \in P}{(H, L, T, P, S, s) \vdash \texttt{set}(\odot, k, v) = \textit{nil}, (H, L, T, P \cup (\odot, k, v), S, s)}$$

$$\text{(MODIFY)}\ \frac{v \in H \cup \textit{Imm} \quad \exists v' (\odot, k, v') \in P}{(H, L, T, P, S, s) \vdash \texttt{set}(\odot, k, v) = \textit{nil}, (H, L, T, P \backslash \{(\odot, k, v')\} \cup \{(\odot, k, v)\}, S, s)}$$

$$\text{(UNSET-1)}\ \frac{\exists (\odot, k, v) \in P}{(H, L, T, P, S, s) \vdash \texttt{unset}(\odot, k, v) = \textit{nil}, (H, L, T, P \backslash \{(\odot, k, v)\}, S, s)}$$

$$\text{(UNSET-2)}\ \frac{\nexists (\odot, k, v) \in P}{(H, L, T, P, S, s) \vdash \texttt{unset}(\odot, k, v) = \textit{nil}, (H, L, T, P, S, s)}$$

$$\text{(CREATE-NODE)}\ \frac{\bullet_n \notin H}{(H, L, T, P, S, \bullet_p :: s) \vdash \texttt{create\_node}(\textit{nil}) = \bullet_n, (H \cup \{\bullet_n\}, L, T, P, S \cup \{(\bullet_p, \bullet_n)\}, \bullet_n :: \bullet_p :: s)}$$

$$\text{(CREATE-BLOCK)}\ \frac{\circ_n \notin H}{(H, L, T, P, S, \bullet_p :: s) \vdash \texttt{create\_block}(\textit{nil}) = \circ_n, (H \cup \{\circ_n\}, L, T, P, S \cup \{(\bullet_p, \circ_n)\}, \bullet_p :: s)}$$

$$\text{(CREATE-ROOT-NODE)}\ \frac{\bullet_n \notin H}{(H, L, T, P, S, []) \vdash \texttt{create\_node}(\textit{nil}) = \bullet_n, (H \cup \{\bullet_n\}, L, T, P, S, \bullet_n :: [])}$$

$$\text{(CLOSE-SCOPE)}\ \frac{}{(H, L, T, P, S, \bullet_p :: s) \vdash \texttt{close}(\textit{nil}) = \textit{nil}, (H, L, T, P, S, s)}$$

$$\text{(CREATE-ROOT-BLOCK)}\ \frac{\circ_n \notin H}{(H, L, T, P, S, []) \vdash \texttt{create\_block}(\textit{nil}) = \circ_n, (H \cup \{\circ_n\}, L, T, P, S, [])}$$

$$\text{(CLOSE-SCOPE)}\ \frac{}{(H, L, T, P, S, \bullet_p :: s) \vdash \texttt{close}(\textit{nil}) = \textit{nil}, (H, L, T, P, S, s)}$$

$$\text{(REOPEN-SCOPE)}\ \frac{\bullet_p \in H \cap \textit{Node}}{(H, L, T, P, S, s) \vdash \texttt{reopen}(\bullet_p) = \textit{nil}, (H, L, T, P, S, \bullet_p :: s)}$$

$$\text{(DETACH-1)}\ \frac{\exists \bullet_p \in H \cap \textit{Node}, \bullet_n \in T(\bullet_p)}{(H, L, T, P, S, s) \vdash \texttt{detach}(\bullet_n) = \textit{nil}, (H, L, T \backslash \{(\bullet_p, l)\} \cup \{(\bullet_p, l - \bullet_n)\}, P), S, s}$$

$$\text{(DETACH-2)}\ \frac{\bullet_n \in H \cap \textit{Node} \quad \textit{Anc}(\bullet_n) = \emptyset}{(H, L, T, P, S, s) \vdash \texttt{detach}(\bullet_n) = \textit{nil}, (H, L, T, P, S, s)}$$

- Implicit copy

$$\text{(ATTACH)}\ \frac{\bullet_p \in H \cap \textit{Node} \quad \bullet_n \in H \cap \textit{Node} \quad \textit{Anc}(\bullet_n) = \emptyset \quad \bullet_n \notin \textit{Anc}(\bullet_p)}{(H, L, T, P, S, s) \vdash \texttt{bind}(\bullet_p, \bullet_n) = \textit{nil}, (H, L, T[\bullet_p arrow \bullet_n :: T(\bullet_p)], P, S, s)}$$

$$\text{(ATTACH-COPY)}\ \frac{\begin{array}{c}\bullet_p \in H \cap \textit{Node} \quad \bullet_n \in H \cap \textit{Node} \quad \textit{Anc}(\bullet_n) \neq \emptyset \vee \bullet_n \in \textit{Anc}(\bullet_p) \\ (H, L, T, P, S, s) \vdash \texttt{copy}(\bullet_n) = \bullet_{n'}, (H', T', P', S', s)\end{array}}{(H, L, T, P, S, s) \vdash \texttt{bind}(\bullet_p, \bullet_n) = \textit{nil}, (H', L', T'[\bullet_p arrow \bullet_{n'} :: T'(\bullet_p)], P', S', s)}$$

## Rule for explicit copy 13

$$\text{(COPY)}\ \frac{\begin{array}{rcll} H' = H & \cup & \{\odot | (\cdot, \odot) \in C\} & \text{where } \textit{rebind}(\odot \in H) = \odot' \text{ if } (\odot, \odot') \in C, \odot \text{ otherwise} \\ L' = L & \cup & \{(\bullet', l) | (\bullet, \bullet') \in C, (\bullet, l) \in L\} & \\ T' = T & \cup & \{(\bullet', l') | (\bullet, \bullet') \in C, l = T(\bullet), l' = \textit{map}(\textit{rebind}, l)\} & \\ P' = P & \cup & \{(\odot', k, v') | (\odot, \odot') \in C, v' = \textit{rebind}(P(\odot, k))\} & \\ S' = S & \cup & \{(\bullet', \odot') | (\bullet, \bullet') \in C, (\odot, \odot') \in C \cup C\} & \end{array}}{(H, L, T, P, S, s) \vdash \texttt{copy}(\bullet_n) = A^{\bullet}(\bullet_n), (H', L', T', P', S', s)}$$

The result is the union of

- The original state
- Duplication of accessible nodes in the scope of the copy target
- Duplication of internal links and preservation of external links, using rebind

The set of copied nodes is computed as follows

- We collect all the candidates to the copy using scope information
  $I = \textit{fix}(\textit{Collect}, \{\bullet_n\}) \,/\, \textit{Collect}(E) = \bigcup_{\bullet \in E} \textit{Desc}(\bullet) \cup \bigcup_{\odot \in E} \{\odot' | (\odot, \odot') \in S\}$
- We restrict to accessible objects
  $R = \textit{fix}(\textit{Restrict}, \{\bullet_n\}) \,/\, \textit{Restrict}(E) = \bigcup_{\bullet \in E} \textit{Desc}(\bullet) \cup \bigcup_{\odot \in E} \{\odot' | (\odot, \cdot, \odot') \in P \wedge \odot' \in I\}$
- We associate original nodes and copies
  $C = \{(\odot, \odot') | \odot \in R, \odot' \notin H \text{ (fresh node/block)}\}$

# Conclusion

## Implementation of *ᶜDOM* 14

Two possibilities :

- Scope of each node stored as a list of pointer to objects
- Each allocation stores a hidden pointer to the last opened node

Our prototype :

- Based on OBrowser, OCaml virtual machine in JavaScript
- Tags each language value with the last opened node
- High level construction specified but not yet implemented
- No unexpected performance cost

## Conclusion 15

We proposed a new document model which

- Enables the use of declarative languages and abstractions in the browser
- Can be implemented over the DOM
- Has a formal specification

What remains to do

- A server side (native) implementation
- Try and integrate *ᶜDOM* into the Ocsigen framework
- Generalize the work to any delimited language structure (eg. objects, modules)?

Work published in Practical Aspects of Declarative Programming 2013