A declarative-friendly API for Web document manipulation

Benjamin Canou, Emmanuel Chailloux, Vincent Balat

Rome, January 21-22, 2013

Introduction

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 for document manipulation
- 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
- Presentation of ^{C}*DOM*, our new document model
- Conclusion and future works

Implicit moves in the DOM

Explanation of the behaviour
Influence on programming and type systems

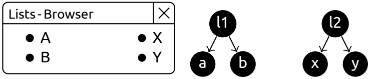
Implicit moves

- The internal representation of the document is a tree
- The DOM is a general, low level graph API
- Actions that would introduce sharing or cycles
  - Are rejected dynamically (exceptions)
  - Perform side effects to preserve the structure : implicit moves

- 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> ...
```

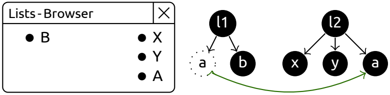
- Rendering and DOM



- We execute the following JavaScript

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

- Resulting in an implicit move



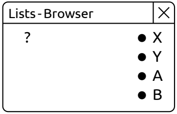
Breaking the validity

With imperative manipulations:

- Using another JavaScript

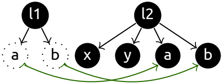
```
var l2 = getElementById ("L2")
var a = getElementById ("A")
var b = getElementById ("B")
  l2.appendChild (a)
  l2.appendChild (b)
```

- We break the validity (empty list)



?

- X
- Y
- A
- B



With purely constructive code:

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

```
(let ((a (<LI> "A"))
      (b (<LI> "B")))
  (<DIV> (<UL> (a))
         (<UL> (a b))))
```

- 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>
```

Summary

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
- Shared instances have to be expanded / duplicated : not so simple

Presentation of our solution

The main idea
Structure of the solution

Implicit copies instead of moves 1 / 2

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

Implicit copies instead of moves 2 / 2

We need a sensible and intuitive convention

- To let the programmer know / decide whether objects belong to a node or not
- In an appropriate way for the high level language / document model

To be as generic as possible:

- We define 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 article:

- We give a glimpse of our work on the high level part for the intuition
- What we present the generic, low level layer: $^{c}\mathit{DOM}$

Overview of the high level part

In ML, we introduce a delimited node definition syntax

- We let the programmer decide whether objects belong to a node or not
- We reuse the familiar notion of lexical scope
- 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 ;;
```

Definition of ${}^{c}\mathit{DOM}$

Structure of the specification 1 / 2

An 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

Specified as follows:

- As set of simply typed, language agnostic primitives
- Formal specification of the document state
- Operational semantics rules (complete spec in the paper)

And a few properties:

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

Structure of the specification 2 / 2

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

Implementations of run-time scope information:

- Scope of each node stored as a list of pointer to objects (as in the spec)
- Each allocation stores a hidden pointer to the last opened node
- In both cases, we need weak references
- The scope stack is managed by the allocator

The API

- 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

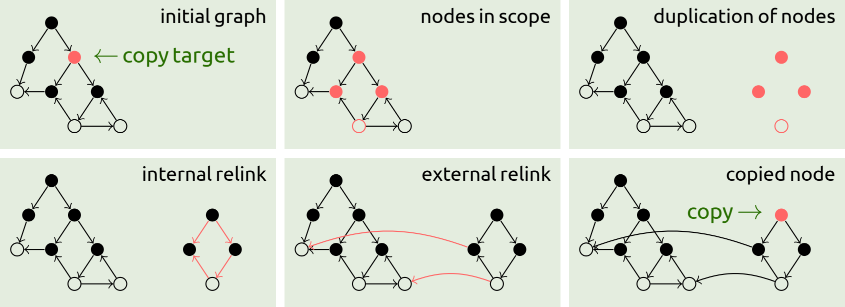
Node `create_node` (*Tag*)	*new, empty node + open its scope*
Nil `close` (*Node*)	*close the scope of a node*
Object `create_block` (*Nil*)	*new, empty block*

- 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*

Copy mechanism

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



Copy algorithm in 3 recursive traversals:

- Collect all the candidates to the copy using scope information
- Restrict to accessible objects
- Duplicate the nodes and create a map from original to copies
- Traverse the original value and perform internal and external relinking using the map

Conclusion

Conclusion, ongoing and future works

We proposed a new document model which fixes the DOM:

- Does not perform unexpected side effects
- Preserves type safety thanks to structure preservation
- Allows the use existing high level systems for XML in the browser
- Has a formal specification

And introduces new possibilities:

- Type checking of imperative manipulations of the Web page
- Explicit copy is a new tool given to the programmer
- Use meta information for other purposes, eg. serialization, migration ?
- Generalize to other delimited language structure (eg. objects, modules) ?

What remains to do:

- A server side (native) implementation
- Try and integrate ^{C}DOM into the Ocsigen framework