A declarative-friendly API for Web document manipulation

Benjamin Canou[1], Emmanuel Chailloux[1], and Vincent Balat[2]

[1] LIP6 - UMR 7606, Université Pierre et Marie Curie, Sorbonne Universités, 4 place Jussieu, 75005 Paris, France
[2] CNRS, PPS UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité, F-75205 Paris, France

Abstract. The Document Object Model (DOM) is the document manipulation API provided to the JavaScript developer by the browser. It allows the programmer to update the currently displayed Web page from a client side script. For this, DOM primitives can create, remove or modify element nodes in the internal tree representation of the document. Interactive documents can be created by attaching event handlers and other auxiliary data to these nodes. The principle is interesting and powerful, and no modern Web development could be possible without it. But the implementation is not satisfactory when seeking predictability and reliability, such as expected with declarative languages or static type systems. Primitives are too generic, and when called in abnormal conditions can either throw exceptions or perform implicit imperative actions. In particular, DOM primitives can conditionally and implicitly move nodes in the document, in a way very difficult to be statically prevented or even detected. In this article, we introduce ^{c}DOM, an alternative document model that performs implicit deep copies instead of moves. By not moving their children implicitly, it preserves the structure of nodes after their creation and between explicit mutations. Side data embedded in the document are also duplicated in a sensible way so that the copies are completely similar in structure to the originals. It thus provides a more usual semantics, over which existing declarative abstractions and type systems can be used in a sound way. [3]

Keywords: document manipulation, Web programming, multi-paradigm

1 Introduction

In most widespread Web programming solutions, the programmer has to master numerous programming languages and environments. At least, she has to know HTML and CSS, the content description languages, some server language to generate pages like PHP or Perl, and the JavaScript programming language

[3] Work partially supported by the French national research agency (ANR), PWD project, grant `ANR-09-EMER-009-01`, and partly performed at the IRILL center for Free Software Research and Innovation in Paris, France

to make Web pages interactive. Moreover, she has to handle the interactions between these languages, a task usually done with very rudimentary techniques, like directly writing client code using strings of the server language. This issue is known in the literature as an *impedance mismatch* problem. Recent Web solutions, from industry (such as Google Web Toolkit or Node.js) as well as research projects (such as Links [4], Ocsigen [1] or HOP [9]), have already tackled this problem, putting a great amount of work to give a uniform solution to program the server and the browser at language level.

However, another impedance mismatch problem remains in these solutions. The document manipulation APIs are different on the server and in the browser, making the programmer work with the same document in very different ways. In the past, this has not really been an issue. The server only produced the document, while the client updated it by inserting new document parts dynamically requested to the server. But in modern Web applications, this separation of roles is not true anymore. The server side may want to perform mutations on the document, while the client side certainly wants to create new content without asking for the server to generate it. Providing the same document manipulation API on both sides has thus become a key point for integrated client-server frameworks.

For untyped, imperative software, such as Node.js which brings JavaScript to the server, this can simply be done by using the DOM on both sides. This indeed provides a uniform API, albeit a very low level one. However, as we detail in Section 2, the unusual semantics of the DOM make it incompatible with declarative languages and advanced static type systems, neither directly as an API, nor even as a low level implementation layer. Several solutions or workarounds have already been tried to enable the use of the DOM in these contexts, but none is completely satisfactory. The solution we present in this article takes another direction by designing an alternative document model. This model, ^{c}*DOM*, is as low level as the DOM so it can be used as a replacement. However, its semantics is much more suited to be used by high level abstractions. One of the main goals is to be able to reuse existing high levels languages and tools for XML to manipulate the document in the browser. We present the intuition behind ^{c}*DOM* in Section 3 and then give its formal description in Section 4, along with a few implementation details. Section 5 then presents the related works, and we finally conclude this article by presenting our future research works around the topic.

2 What is wrong with the DOM

This section explains why using the DOM for document manipulation is not an option both for declarative programming and static typing. Section 2.1 presents our main source of concern: implicit moves. It features an example that behaves counter-intuitively from the point-of-view of a declarative programmer. Section 2.2 presents the problems indirectly introduced by implicit moves. It explains why implicit moves make some type checking problems too difficult for existing static type systems, how they hinder even purely functional document construction, and how they impact client-server programming. Section 2.3 then makes a quick tour of existing solutions and workarounds to these problems.

2.1 Implicit moves

In specific situations, the DOM can implicitly move nodes in the document tree. Let us examine an example which leads to such an implicit move in order to understand why and when. An initial page consists of two lists containing two items each. The elements of interest have been marked by hand with *id* attributes in order to be able to retrieve the corresponding DOM nodes from JavaScript (the standard technique). Figure 1 shows the HTML code, the rendering and a visualization of the DOM tree. Within this page, we run JavaScript code that calls DOM primitives to take the first element of the first list, and append it to the second list. The effect on the DOM is indeed the insertion of the item in the second list, but at the same time its deletion from the first list. Figure 2 shows the code and the outcome. This behaviour may appear counter-intuitive to the programmer unfamiliar with the DOM. She asked for an append operation, not a move. As a result of implicit moves, the outcome of a series of DOM operations can depend a lot on the initial state and be hard to predict.

```
... <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> ...
```

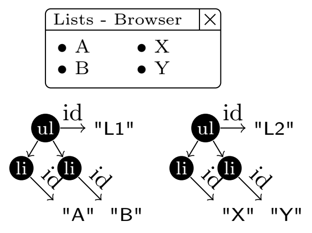


Fig. 1. A simple page

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

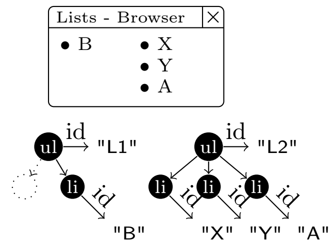


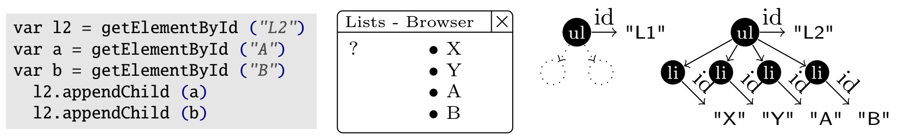
Fig. 2. An example of implicit move

Implicit moves are introduced to compensate the fact that the set of DOM primitives does not exactly fit the internal structure of the document. The memory representation of the page has to be a tree. The reason is obvious: page elements are bound to graphical objects, so cycles or sharing in the structure would not make sense. However, the set of primitives describes imperative operations on a general graph structure. A DOM primitive application that would introduce sharing or cycles in the structure is thus given an alternative behaviour that preserves the tree shape.

2.2 Side effects of implicit moves

To be correctly handled by browsers, documents have to conform to precise formats. Specific, strongly typed XML processing languages such as CDuce [3] make it possible to ensure the validity of generated documents on the server. But for DOM intensive applications, verifying the validity of the initial Web page is not enough. It is often just a stub, enriched as data arrives from independent HTTP requests. It ensues that page modifications also have to be proven preserving the grammar.

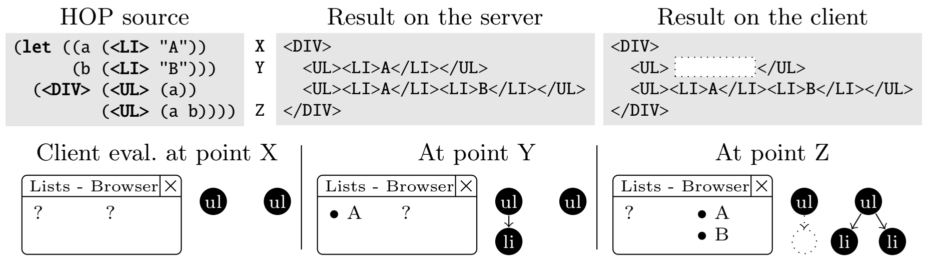
Breaking validity during manipulations Unfortunately, static checking of DOM operations is difficult, mainly because of implicit moves (which are not present in high level XML APIs). For instance, figure 3 shows a program which is well typed at first glance, since it takes references to *li* elements and adds them to an *ul* element, which is correct according to the XHTML grammar. But the outcome of executing this program with the initial (valid) Web page of figure 1 is a broken Web page, since the first list is now empty, and thus not well typed anymore. Of course this minimal example is trivial and is presented only to exhibit the problem: a DOM manipulation on one node can dynamically break the validity of another node.



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

Fig. 3. A validity breaking implicit move

Breaking validity during construction It is even possible to obtain an invalid result from a purely constructive code that would lead to a correct result using an XML language. The reason is that XML manipulation APIs allow sharing, whereas the DOM forbids sharing using implicit moves. Figure 4 shows an example HOP program, its evaluation on the server (where an XML representation that allows sharing is used) and on the client (where the back-end is the DOM). On the server, the result is the one expected by the programmer while on the client, an implicit move occurs and the first list ends up empty. Ensuring that these cases do not occur is left to the programmer. As a result, using the DOM as a back-end for high-level abstractions is not a reliable option.



HOP source

```
(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>
```

Fig. 4. Well-typed code leading to an invalid result

Influence on server document models In integrated client-server applications, the task is not only to produce a valid document. The server also has to produce a document that will be manipulated by a client program in a valid way. In this respect, existing, server side only document models are insufficiently expressive. The problem resides in the transition between the server representation and the DOM. The technique used is always the same: nodes referenced by the server side are retrieved by the client side by searching the DOM tree for unique identifiers inserted in the XML (manually of automatically). If the server representation allows sharing, which is the case for most high level solutions, a shared node in this representation will be expanded to duplicate XML elements. Thus, any identifier it may contain will be duplicated as well, leading to undefined behaviour on the client.

2.3 Existing solutions to handle implicit moves

Of course, this problem is not new. Several solutions have been introduced by Web frameworks as well as research works. But to our knowledge, none of them offer as much versatility as the one we propose and most of them have non negligible drawbacks.

Client-server JavaScript solutions (for instance using Node.js) can use the DOM on both sides. But as we already explained, this is only an option in such untyped, imperative contexts.

High level, language based client-server frameworks (eg. HOP, Ocsigen, Links, OPA, Ur/Web) use an intermediate representation as we explained earlier. This solution has many advantages for document construction, but when it comes to document mutations, it is not better than using the DOM. The programmer has to make sure not to introduce sharing in the documents she builds, otherwise nasty side effects (including implicit moves) may occur later. Moreover, the task is actually as difficult as ensuring that no implicit move occurs with the DOM.

More mainstream frameworks hide the document structure and rely on prebuilt components instead. User code is mostly glue code, written either in server code (eg. Ruby on Rails, django) or in JavaScript (eg. ExtJS, Dojo). The main restriction is that the programmer has to compose its pages only of existing components of some framework. Frameworks are often incompatible, and writing a custom component means learning hackish internal details.

Research solutions already exist to prevent implicit moves by programs using the DOM. However, their integration to existing mainstream solutions is difficult due to the very advanced techniques used. Related research works will be detailed in Section 5.

3 An implicitly copying document model

All the existing solutions we just presented try to prevent situations which lead to implicit moves. Our solution takes a different path. ^{c}DOM is an alternative document model, as imperative and low level as the DOM but without implicit moves. Instead, $^{c}\mathit{DOM}$ performs implicit copies. It leaves the original node in place, and inserts a copy of it at destination.

3.1 Deep copies and auxiliary data

To preserve document validity, deep (as opposed to shallow) copies have to be performed so the grammatical structure of original and copied nodes are the same. With existing solutions, the programmer can already detect when an implicit move can occur and replace it manually by a copy. $^{c}\mathit{DOM}$ simply makes this operation systematic. Recursively copying children nodes is not difficult and the DOM primitive *cloneNode* already does that very operation. But for interactive documents, such a recursive copy of nodes is not enough. To preserve the behaviour of nodes, associated auxiliary data have to be copied along with the nodes, in particular event handlers. With existing solutions, this task is manual and non trivial. It requires the programmer to make sensible decisions about what to copy, what not to and what to share with the original node. If auxiliary data are structured language values, the programmer has to decide how deep the copy has to be. Moreover, in the case of static typing, the copy operation has to preserve the types of original auxiliary data. For all these reasons, it is not possible to provide a generic deep copy algorithm for the DOM. Figure 5 shows the definition of a node and two examples of non trivial decisions to be made during a copy of this node.

```
var cpt = new Counter (0);
var node = document.createElement ("button");
node.appendChild (document.createTextNode("0"));
node.onclick = function () {
  cpt.incr (); ←——— Should a copy of node use the same counter cpt or a copy?
  var lbl = cpt.stringValue();
  node .appendChild (document.createTextNode(lbl));
}   ↑_________________ Should a copy of node modify itself or the original?
```

Fig. 5. Different options for the deep copy of a node

3.2 A sensible deep copy algorithm

In the previous example, the lack of convention made it hard to decide whether side data had to be copied or not. Providing a usable copy mechanism thus means providing such a convention, preferably an intuitive one. The major contribution of this work is thus the convention we propose, which is as follows: (1) let the programmer decide whether side data are associated to some node or not, and (2) reuse the familiar notion of lexical scope to materialize this choice. In practice, the idea is to introduce a clearly delimited syntactic construction for node definition, and use it to delimit the set of values to be copied along. Figure 6 gives an example written in such a high level (here ML based) language. In this example, if a copy of the node occurs, in both cases the callback will be copied along and act on the copied node rather than the original. However, in one case it will use a shared counter and in the other a local copy, depending on the location of its definition.

With shared reference

```
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 ;;
```

With local references

```
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 ;;
```

Fig. 6. Self modifying graphical counter in an ML frontend to ^{c}*DOM*

This mechanism is more predictable, not only by the programmer but also by tools, in particular type systems. With implicit copies, the implicit mutation of nodes content is now gone. The only times when a node is modified are its creation and explicit modification. These cases can be handled by type checking the new assigned content, for instance with existing type systems for XML. The only additional restriction is that the node should not be used until it is completely built, so that a copy of an incomplete node cannot occur.

We chose not to limit our solution to a specific high level language, but to build a foundation on top of which various languages and abstractions could be built or ported. The solution we propose is to add meta-information to nodes directly in the low-level document model. This information is used as an oracle for a generic copy algorithm to decide which objects are to be copied along with the node. As we just explained, these meta-information can be used to maintain lexical scoping information at run-time. However, ^{c}*DOM*'s meta information storage is actually flexible enough for meta-information to be used in other ways, for instance to be adapted to language not equipped with a clear lexical scoping.

4 Formal specification of ^{c}*DOM*

As the DOM, ^{c}*DOM* takes the form of a language independent API. However, ^{c}*DOM* is specified more formally by an operational semantics. This section first gives precise yet informal definitions of the concepts and then the specification.

4.1 Main concepts

- *Document* The main concept we are formalizing is the *document* as used in the Web (eg. XML, DOM). A document assembles *nodes* that can represent textual content, graphical and semantic elements in a hierarchical structure.
- *Node* A node can have *children* nodes, and can have (at most one) *parent* node. It has a *tag* which defines its role in the document. Several nodes can

have the same tag in a document. This role is not defined by the document itself but by the program interpreting it (for instance, a Web browser will render a bullet list when it encounters a *ul* tag). The formalism presented stays at the DOM level in this sense: it is a uniform representation and does not bear any grammar notion.

– *Values* To handle textual content and programmable interactions, document nodes are enriched with side data. *Values* is the term we use to designate both nodes and side data. We distinguish *immediate* data such as integers and strings from structured data that we call *blocks* (in JavaScript, blocks designate language objects, including functions). The relation between these types is the following: $\mathit{Value} = \mathit{Imm} \cup \mathit{Object}$, $\mathit{Object} = \mathit{Block} \cup \mathit{Node}$.
– *Properties* Nodes and side data are linked using *properties*: associations between *objects* and *values* labeled with *keys*. Unlike nesting, properties can lead to sharing and cycles in the document.
– *Imperative Document* The *document* notion we just defined is inherently static, and thus not appropriate to formalize the DOM. We define the notion of *imperative document* combining a *document* as previously defined that we call the *state* with a set of *primitives* to manipulate it.
– *Primitives* To implement this separation, ${}^{C}\mathit{DOM}$ is specified as a set of *primitives*, an API, much as the DOM. They take parameters and return results, which are *values* as specified earlier.

4.2 Parameters

In the previous section, we described the main concepts and associated types provided by our formal model. These definitions are made more flexible by defining some of the notions as parameters, so they are not fixed by the model but are to be instantiated specifically for each implementation.

– *Tag* The set of possible node tags. There are no constraints on this parameter for the semantics to be sound but a specific implementation may add some.
– *Imm* The (unrestricted) domain of immediate values.
– *Key* The domain of object property names. It has to be enumerable and provided with a total order. In practice, keys have to be immutable.
– *Nil* The type of unimportant values. In this formalism, Nil is not an implicit subtype of everything. Types that contain Nil will be written as such.
– *Int* The representation of integers. The formalism relies on the mathematical definition, but in practice, there is no chance for a document to contain a node with a number of children that would trigger computer arithmetic overflows, so the approximation is reasonable.
– *Enum*(S) Some primitives return not only one but a collection of results. *Enum*(S) is the representation of collections of elements of a type S. In the semantics, the transition between mathematical sets and concrete collections is exhibited by the use of the function $\mathit{enum} : \mathcal{P}(S) \rightarrow \mathit{Enum}(S)$.

4.3 Document state

The document state is specified in ^{C}DOM by a tuple (H, L, T, P, S, s). Letters are mnemonics for Heap, Labels, Tree, Properties, Scopes and Stack. The first four components (H, L, T, P) describe the document structure while the last two (S, s) describe the dynamic scoping information.

- $H \subseteq Node \cup Block$ 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.
- $S \subseteq Node \times Object$ records for each nodes the objects under its scope.
- $s \in List(Node)$ represents the stack of currently opened scopes.

We intentionally chose a simple mathematical structure to ease implementation, and be close to data structures. But this structure is not precise enough to express the document structure. We thus restrict it using the following *well-formed* predicate. A notable point is that this predicate is only useful at specification level and is transparent to the implementer: well-formedness is preserved by definition of the primitives. The implementer only has to correctly map the specification to her data structure and the body of her primitives.

Definition 1. *A tuple (H, L, T, P, S, s) is a well-formed ^{C}DOM state if and only if (1) L maps each node in H to a unique tag (2) T is a forest (no sharing, no cycles) over $H \cap Node$ (3) T and P only reference nodes present in H (4) P only references blocks present in H (5) An object can be in the scope of only one node in S (6) No cyclic scope chain exist in S.*

Notations To increase readability, in the following formulas and figures, $\bullet$ means a node, $\circ$ a block and ◐ an object. Labeled versions are used when disambiguation is required (eg. $\bullet_x$, $\circ_y$). We also define operators to compute the descendants and ancestors of a node.

$$Desc(\bullet) = \bigcup_{\bullet' \in T(\bullet)} (\{\bullet'\} \cup Desc(\bullet'))$$
$$Anc(\bullet) = \{\bullet'\} \cup Anc(\bullet') \text{ if } \exists \bullet', \bullet \in T(\bullet'), \emptyset \text{ otherwise}$$

4.4 API

The following list gives the complete ^{C}DOM API, the parameters and result types in the form *return type* `primitive` *(types of parameters)*. ^{C}DOM primitives are divided into two main subsets: accessing (reading) primitives and modifying (writing) primitives.

- *Int* `children` (*Node*)
- *Enum*(*Node*) `roots` (*Nil*)
- *Value* + *Nil* `get` (*Object*, *Key*)
- *Node* `create_node` (*Tag*)
- *Nil* `close` (*Node*)
- *Nil* `detach` (*Node*)
- *Nil* `bind` (*Node*, *Node*)
- *Nil* `unset` (*Object*, *Key*)
- *Node* + *Nil* `child` (*Node*, *Int*)
- *Enum*(*Key*) `properties` (*Object*)
- *Tag* `tag` (*Node*)
- *Object* `create_block` (*Nil*)
- *Nil* `reopen` (*Node*)
- *Node* `copy` (*Node*)
- *Nil* `set` (*Object*, *Key*, *Value*)

Semantic rules The behaviour of each primitive is described by a set of (for some only one) semantic rule(s). Each rule is of the form

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

reading: given arguments $(a_0,\cdots,a_n)$ and an initial state S, if the *conditions* are verified, the primitive `prim` can be applied, and the rule (RULE) can be elected to describe the behaviour of this application. If so, its return value is r, and the original state is transformed into the new state S'.

Accessing primitives Figure 7 gives the semantics of primitives which are only meant to read the document state from the host language, without modifying it. To browse the document tree, `roots` gives the root nodes (more than one document root can be present, for instance any newly created node is considered a root), `children` and `child` allow to browse the structure by respectively giving the number of children and the n^{th} child of a node. The set of properties defined by a given node is obtained with `properties`, and `get` gives the value of a given property. The tag of a node is given by `tag`.

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

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

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

$$\frac{\textbf{(PROPERTIES)}\qquad \oslash \in H}{S\ \vdash\ \texttt{properties}(\oslash) = \mathit{enum}(\{k | (\oslash, k, v) \in P\}), S}$$

$$\frac{\textbf{(GET)}\quad \exists(\oslash, k, v) \in P}{S\ \vdash\ \texttt{get}(\oslash, k) = v, S} \qquad \frac{\textbf{(GET-UNBOUND)}\qquad \nexists(\oslash, k, v) \in P}{S\ \vdash\ \texttt{get}(\oslash, k) = \mathit{nil}, S}$$

Fig. 7. Semantics of accessing primitives

Modifying primitives Figure 8 gives the semantics of primitives that modify the document state. For block related primitives, `create_block` allocates a new, empty one, `set` either creates or assigns a property depending on its preexistence and `unset` removes a property. For node related primitives, `create_node` allocates a fresh one, `detach` removes the link between a node and its parent, and `bind` links a node to a parent. Two rules describe the evaluation of the later: either (1) the node is simply attached to its new parent if it is a root and if the new link does not create a cyclic chain in T, or (2) a deep copy of the node is performed by delegation to the explicit `copy` primitive and the result is attached to the parent.

Scope information When a new node is allocated, its scope is automatically opened on the scope stack. Scopes are explicitly closed, using the `close` primitive. We also added a `reopen` primitive to push again on the scope stack a node whose scope has already been closed. This primitive may or may not be necessary, depending on the high level primitives given to the programmer. For instance,

$$\dfrac{\textbf{(SET)}\quad v \in H \cup Imm \qquad k \in Key \qquad \LEFTcircle \in H \qquad \nexists v', (\LEFTcircle, k, v') \in P}{(H,L,T,P,S,s) \;\vdash\; \texttt{set}(\LEFTcircle, k, v) = nil, (H,L,T,P \cup (\LEFTcircle,k,v), S, s)}$$

$$\dfrac{\textbf{(MODIFY)}\qquad v \in H \cup Imm \qquad \exists v' \left(\LEFTcircle, k, v'\right) \in P}{(H,L,T,P,S,s) \;\vdash\; \texttt{set}(\LEFTcircle,k,v) = nil, \left(H,L,T,P \backslash \{(\LEFTcircle,k,v')\} \cup \{(\LEFTcircle,k,v)\}, S, s\right)}$$

$$\dfrac{\textbf{(UNSET-1)}\qquad \exists\,(\LEFTcircle,k,v) \in P}{(H,L,T,P,S,s) \;\vdash\; \texttt{unset}(\LEFTcircle,k,v) = nil, (H,L,T,P \backslash \{(\LEFTcircle,k,v)\}, S, s)}$$

$$\dfrac{\textbf{(UNSET-2)}\qquad \nexists\,(\LEFTcircle,k,v) \in P}{(H,L,T,P,S,s) \;\vdash\; \texttt{unset}(\LEFTcircle,k,v) = nil, (H,L,T,P,S,s)}$$

$$\dfrac{\textbf{(CREATE}^\bullet\textbf{)}\qquad \bullet_n \notin H}{(H,L,T,P,S,\bullet_p :: s) \;\vdash\; \texttt{create\_node}(nil) = \bullet_n, (H \cup \{\bullet_n\}, L, T, P, S \cup \{(\bullet_p, \bullet_n)\}, \bullet_n :: \bullet_p :: s)}$$

$$\dfrac{\textbf{(CREATE}^\circ\textbf{)}\qquad \circ_n \notin H}{(H,L,T,P,S,\bullet_p :: s) \;\vdash\; \texttt{create\_block}(nil) = \circ_n, (H \cup \{\circ_n\}, L, T, P, S \cup \{(\bullet_p, \circ_n)\}, \bullet_p :: s)}$$

$$\dfrac{\textbf{(CREATE-ROOT}^\bullet\textbf{)}\qquad \bullet_n \notin H}{(H,L,T,P,S,[\,]) \;\vdash\; \texttt{create\_node}(nil) = \bullet_n, (H \cup \{\bullet_n\}, L, T, P, S, \bullet_n :: [\,])}$$

$$\dfrac{\textbf{(CLOSE-SCOPE)}}{(H,L,T,P,S,\bullet_p :: s) \;\vdash\; \texttt{close}(nil) = nil, (H,L,T,P,S,s)}$$

$$\dfrac{\textbf{(CREATE-ROOT}^\circ\textbf{)}\qquad \circ_n \notin H}{(H,L,T,P,S,[\,]) \;\vdash\; \texttt{create\_block}(nil) = \circ_n, (H \cup \{\circ_n\}, L, T, P, S, [\,])}$$

$$\dfrac{\textbf{(REOPEN-SCOPE)}\qquad \bullet_p \in H \cap Node}{(H,L,T,P,S,s) \;\vdash\; \texttt{reopen}(\bullet_p) = nil, (H,L,T,P,S,\bullet_p :: s)}$$

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

$$\dfrac{\textbf{(DETACH-2)}\qquad \bullet_n \in H \cap Node \qquad Anc(\bullet_n) = \emptyset}{(H,L,T,P,S,s) \;\vdash\; \texttt{detach}(\bullet_n) = nil, (H,L,T,P,S,s)}$$

$$\dfrac{\textbf{(ATTACH)}\qquad \bullet_p \in H \cap Node \qquad \bullet_n \in H \cap Node \qquad Anc(\bullet_n) = \emptyset \qquad \bullet_n \notin Anc(\bullet_p)}{(H,L,T,P,S,s) \;\vdash\; \texttt{bind}(\bullet_p, \bullet_n) = nil, (H,L,T\left[\bullet_p \to \bullet_n :: T(\bullet_p)\right], P, S, s)}$$

$$\dfrac{\begin{array}{c}\bullet_p \in H \cap Node \qquad \bullet_n \in H \cap Node \qquad Anc(\bullet_n) \neq \emptyset \vee \bullet_n \in Anc(\bullet_p) \\ \textbf{(ATTACH-COPY)}\qquad (H,L,T,P,S,s) \;\vdash\; \texttt{copy}(\bullet_n) = \bullet_{n'}, \left(H', T', P', S', s\right)\end{array}}{(H,L,T,P,S,s) \;\vdash\; \texttt{bind}(\bullet_p, \bullet_n) = nil, \left(H', L', T'\left[\bullet_p \to \bullet_{n'} :: T'(\bullet_p)\right], P', S', s\right)}$$

Fig. 8. Semantics of modifying primitives

in an object oriented language, it may be sensible to consider a method call on a node as within its scope. Every new object alocated by one of the `create` primitives is associated in S to the top node present in the scope stack s. If the scope stack is empty, the new object is considered not associated to any node.

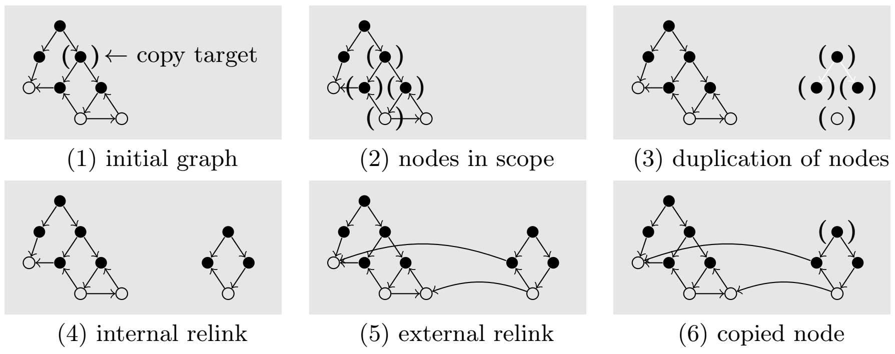
Copy ^{C}DOM provides an explicit `copy` primitive which takes a node $\bullet$ and returns its deep copy $\bullet'$. Descendent nodes are duplicated unconditionally so that no sharing or cycle can occur, and blocks are copied or not depending on scope information. The links between nodes are copied as well, so that the duplicated value has the same memory shape than the original. We will not elaborate on that matter which is a bit out of scope, but as explained earlier, the idea is of course to ensure that both can be considered of the same type, so that the model is usable with strongly typed languages. Let us start with an intuitive description, using the graphical example of figure 10. In (1) and (2) The programmer calls `copy` on a node $\bullet$. The set of objects to copy is composed of all the objects which are reachable from $\bullet$ using both the tree structure or

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

$$
\begin{array}{l}
\text{with } C = \{(\oslash, \oslash') | \oslash \in R, \oslash' \notin H \text{ (fresh node/block)}\} \\
\text{where } R = \mathit{fix}(Restrict, \{\bullet_n\}) \ / \ Restrict(E) = \bigcup_{\bullet \in E} Desc(\bullet) \cup \bigcup_{\oslash \in E} \{\oslash' | (\oslash, \cdot, \oslash') \in P \wedge \oslash' \in I\} \\
\text{and } I = \mathit{fix}(Collect, \{\bullet_n\}) \ / \ Collect(E) = \bigcup_{\bullet \in E} Desc(\bullet) \cup \bigcup_{\oslash \in E} \{\oslash' | (\oslash, \oslash') \in S\}
\end{array}
$$

Fig. 9. Semantics of ^{C}DOM copy operation

properties, and scope information. In (3) Selected nodes are copied by creating fresh nodes in H. In (4) and (5) The parenting links between original nodes are replicated between the duplicates, so that the forest structures of the two groups are the same. All the properties of original blocks are replicated, and the associated values are as follows. If the value is a duplicated node or block its copy is used. Otherwise the value is used as is. In the end (6), all the objects reachable from and in the scope of $\bullet$ are duplicated, the internal links between duplicated objects reflect the structure of the originals, and external links are duplicated as is. Figure 9 gives the formal semantics of the `copy` primitive. The resulting state is the original state augmented with the objects and links resulting from the copy. For this, the rule premises involve a set C of associations between copied objects and their copies. The specification of C is decomposed into the following three steps. First, we collect the set I of all the objects which are descendants or under the scope of $\bullet$, taking care of potential nested scopes. The *Collect* function describes one step of the traversal and its iteration to a fix-point gives the complete collection. We then extract from I the subset R of objects which are reachable from $\bullet$ through the document tree or properties. Finally, C associates original objects to fresh copies.



(1) initial graph (2) nodes in scope (3) duplication of nodes

(4) internal relink (5) external relink (6) copied node

Fig. 10. Illustrated example of a copy operation

4.5 Consistency

To ensure that the API specification is sound and actually corresponds to our needs of modeling the behaviour of the DOM, we have to state that it is deterministic (in other words that it does not bypasses DOM behaviour traits by introducting indeterminism) and that it preserves the well formedness throughout primitive applications, in particular that the copy operation preserves a DOM-like structure.

Definition 2. *A primitive application is deterministic if at most one rule can be selected to describe it.*

Theorem 41 (Determinism) *For one initial state and choice of arguments, at most one rule can be elected to describe the behaviour of a given primitive application.*

Proof. *For each primitive, by showing that any pair of associated rules have mutually exclusive conditions.* □

Theorem 42 (Well-formedness preservation) *A well defined application in a well formed initial state results in a well-formed final state.*

Proof. *For most rules, simply by observing that the required conditions are a sufficient subset of the well-formed predicate clauses. The difficulty resides in the copy operation. We have to prove that all the clauses of the validity predicate are verified over the final state of the copy operation. For this, we observe that every component X' of the resulting state is the union of the original component X and a new set X^+, and that X and X^+ are always disjoint. The union of two forests over disjoint sets of nodes being also a forest, H and H^+ being disjoint, and T^+ being a forest over H^+ (because it is a copy of a subforest of T over H), we have that T' is also a forest. The same reasoning can be used to show that the structural restrictions over S' (no sharing and no cycles) are respected. Finally, a proof that the added properties only reference existing objects is obtained by definition of the rebind function, used to give values to properties in P' using only values of H and H^+.* □

We have proven that the copy operation does not break the model, now we have to prove that it is indeed useful. For this, we define a notion of similarity of structure between two nodes, and prove that the copy operation preserves the structure.

Definition 3. *Two nodes are structurally similar iff (1) they have the same tag, (2) they have the same number of children, and their children are structurally similar pairwise, and (3) They have the same set of properties, and the associated values are structurally similar pairwise. Two objects are the same if they have the same set of properties, and the associated values are structurally similar pairwise. Two immediate values are similar if they are equal.*

Theorem 43 (Structure preservation) *The node resulting of a copy operation is structurally similar to the the original.*

Proof. *By definition of the copy operation, we have directly the respect of tags, number of children and set of properties for all objects duplicated in the copy. It remains to prove that children and properties' values are similar pairwise. For each of these pairs (v, v') where v is the original and v' the duplicate, by definition of rebind, either v' is v or v' is a copy of v using the same copy definition. For the first case, we use the fact that v is structurally similar to itself. The second case is then proven by coinduction.* □

4.6 Implementing ^{C}DOM

The difficulty of implementing ^{C}DOM comes from scope information, which cannot remain static. It has to be managed at run time. We wrote two (non distributed) research prototypes using two techniques to store the scope information. This section discusses these possibilities.

The first possibility is to store a list in each allocated node, initially empty and dynamically filled with pointers to all the objects allocated within the node's scope. The copy algorithm can remain close to the specification: (1) build transitively the set of objects in the scope starting from the root to copy, (2) traverse the graph, duplicating encountered objects and links, memoizing already copied objects to respect sharing and cycles, and (3) stop following links when they point out of the set built in the first step. The memory overhead is very localized, implying no memory overhead on programs which do not perform document manipulations. By dynamically switching the allocator when not in the scope of any node, there can also be no performance overcost for such programs. Implementing this methods requires an advanced memory mechanism such as weak references or a node specific garbage collection in order not to consider alive forever any value allocated within the scope.

The second possibility is to store in each allocated object a backpointer to the node whose scope it belongs to. The algorithm is a little more complex here, because one cannot easily compute the objects in the scope of a node, apart from traversing the whole memory graph. The method is to build the deep copy by steps, maintaining a set of already copied nodes. At each step, (1) traverse the leaves of the already copied subtree and duplicate nodes and blocks backpointing to an already copied subtree, (2) traverse again the subtree, update pointers considered external at the previous step but now pointing to copied objects, redirecting them to the copies, and (3) iterate until a fix point is reached. For this method too, obtaining exact memory collection is doable only with weak pointers, but the memory leak is much more reasonable. It only arises when a node local value is put in a global reference, and not for any local value. It is thus the technique to choose to implement ^{C}DOM over current JavaScript implementations. With both methods, having a memory exact JavaScript implementation of ^{C}DOM over the DOM implies writing a garbage collection helper function, which browses the document regularly and unlinks unused objects using scope information, enabling the next JavaScript collection to actually delete them. Anyway, this will not remain a major concern for long. Weak references are already present in some browsers and are planned for a forthcoming ECMAScript specification.

5 Related works

Works on DOM calls verification The most advanced theoretical work has been led by Peter Thiemann [10], who proposed to integrate an ad-hoc type system into a general purpose language to check DOM calls in order to refuse statically programs which could result in implicit moves. In the same vein, there have been works on automatic tests generation to reject erroneous DOM transactions [7]. This approach is indeed a possible way to solve the problem of implicit moves, and gives direct solutions to the theoretical problems explained in section 2. However, we have two main concerns, which led us to propose our alternative approach.(1) This kind of checks cannot be directly encoded in type systems or tools available in general purpose languages. Moreover, the types or test cases to produce are complex, so these works rely on automatic solvers. Both these aspects imply practical difficulties to obtain good integration to languages and environments, in particular the difficulty to produce useful error messages and debugging possibilities. (2) This solution rejects programs that appear intuitively correct to the declarative programmer and are accepted by advanced XML functional languages such as CDuce. We thus chose to orient our solution on an alternative document model which accepts such programs.

Works on DOM specification There have been several efforts to formalize the different components of the Web browser. We can cite the formal specification of the now defunct JavaScript 2.0 [8], a minimal formal model of JavaScript [6] or closer to our work a semantics of DOM primitives [5]. We shall not elaborate on these works, because we take an alternative approach: the formal model we develop in this paper is a simplification of the DOM.

Deep copy of DOM nodes Libraries such as jQuery define smart copy operations able to duplicate auxiliary data but only to a limited extent. In particular, event handlers are cloned but their environment is copied only in a shallow way. Hence, the copied node will react to events, but the associated action has a good chance to be performed on the original node instead of the duplicate. It is possible to work around this behaviour by being very careful about what ends up in the environment of the event handler closure. This means having a great knowledge of the language and writing trickier code, such as flattening all the environment by hand in the DOM node, so a shallow copy will suffice, assuming that the event code only accesses its environment in an indirect way through the node.

6 Conclusion, ongoing and future works

This article has presented ^{c}DOM, an alternative to the DOM. The implicit moves of the DOM are replaced by implicit deep copies that take into account auxiliary data, including event handlers. As a result, ${}^{c}DOM$ is more suited than the DOM for contexts in which predictability is important such as declarative languages or static type systems. In particular, existing functional XML languages and type systems can be ported without major modifications to the browser.

Of course, we want to prove our approach correct and usable by experimentation. For this, we have specified [2] and are currently implementing an ML-based language on top of ${}^{c}DOM$ (the one shown in one of the examples), which can run over JavaScript and the DOM, and brings static typing of document manipulations.

References

1. Balat, V., Chambart, P., Henry, G.: Client-server Web applications with Ocsigen. In: WWW2012 dev track proceedings, Lyon, France (April 2012) 1–4
2. Benjamin, C.: Programmation Web Typée (Typed Web Programming). PhD thesis, Université Pierre et Marie Curie (2011) Available at http://www.pps.jussieu.fr/~canou/these.pdf (in French).
3. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-Centric General-Purpose Language. In: 8th International Conference on Functional Programming (ICFP'03), New York, NY, USA, ACM (2003) 51–63
4. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web Programming Without Tiers. In: 5th International Symposium on Formal Methods for Components and Objects (FMCO'06), Springer-Verlag (2006) 266–296
5. Gardner, P.A., Smith, G.D., Wheelhouse, M.J., Zarfaty, U.D.: Local Hoare reasoning about DOM. In: 27th Symposium on Principles of Database Systems (PODS'08), New York, NY, USA, ACM (2008) 261–270
6. Guha, A., Saftoiu, C., Krishnamurthi, S.: The Essence of JavaScript. In D'Hondt, T., ed.: 24th European Conference on Object-Oriented Programming (ECOOP'10). Volume 6183 of Lecture Notes in Computer Science., Springer (2010) 126–150
7. Heidegger, P., Bieniusa, A., Thiemann, P.: DOM Transactions for Testing JavaScript. In Bottaci, L., Fraser, G., eds.: 5th International Academic and Industrial Conference on Testing – Practice and Research Techniques (TAIC PART'10). Volume 6303 of Lecture Notes in Computer Science., Springer (2010) 211–214
8. Herman, D., Flanagan, C.: Status report: specifying javascript with ML. In: Workshop on ML (ML'07), New York, NY, USA, ACM (2007) 47–52
9. Serrano, M., Gallesio, E., Loitsch, F.: Hop, a language for programming the web 2.0. In: 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06), New York, NY, USA, ACM (2006) 975–985
10. Thiemann, P.: A Type Safe DOM API. In Bierman, G., Koch, C., eds.: 10th International Symposium on Database Programming Languages (DBPL'05). Volume 3774 of Lecture Notes in Computer Science., Springer (2005) 169–183