Types and Analysis for Scripting Languages
(Part 4: A Type-Safe DOM API)

Peter Thiemann

Universität Freiburg, Germany

Mini Course, Tallinn, Estland, 25.-27.2.2008

Introduction
Programming Model
Invariants
Formal model
Conclusion
Introduction

What is DOM?

- Document Object Model
- W3C recommendation: DOM Level 3
  http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/
- Statement of purpose

  ... a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of [XML] documents.
Introduction

Where is DOM?

- Implementations for Java, JavaScript, Python, Perl, C#, Fortran, Ada, ...
- Every Web browser (through JavaScript)
- Other applications: Mozilla-based, OpenOffice, XMetaL
- Other specifications: SVG
Introduction
The DOM programming model

- XML document represented by graph
Introduction

The DOM programming model

- XML document represented by graph
- Node types characterized by hierarchy of IDL interfaces
- **Node with subtypes**
Introduction

The DOM programming model

- XML document represented by graph
- Node types characterized by hierarchy of IDL interfaces
- Node with subtypes

Manipulation not straightforward
- Node creation using factory pattern
- Methods to maintain the graph
Introduction

What does a typical structure look like?
What does DOM code look like?

```java
int nr = ...;
Document doc = ...;
Element result = doc.createElement("span");
Attr at = doc.createAttribute("id");
at.value = "draw" + nr;
result.setAttributeNode (at);
```
Introduction

What happens underneath?
DOM’s invariants

- DOM maintains more than the obvious structure
DOM’s invariants

- DOM maintains more than the obvious structure
- Additional pointers must obey invariants
  - Linked nodes must not belong to different documents
  - Some combinations of parent and child nodes types are rejected
- Nodes must form a tree structure:
  - A node must not have more than one parent/owner
  - The graph must not be cyclic
- Violations give rise to run-time errors
DOM’s invariants

- DOM maintains more than the obvious structure
- Additional pointers must obey invariants
  - Linked nodes must not belong to different documents
  - Some combinations of parent and child nodes types are rejected
  - Nodes must form a tree structure:
    - A node must not have more than one parent/owner
    - The graph must not be cyclic
- Violations give rise to run-time errors
Goal of this work

- Reflect invariants in the type structure of the DOM interface
- Guarantee absence of run-time errors by type soundness
Illegal DOM manipulation
Attribute ownership

```java
void highlight (Document doc,
                Element el1,
                Element el2) {
    Attr attr = doc.createAttribute ("class");
    attr.value = "highlight";
    el1.setAttributeNode (attr);
    el2.setAttributeNode (attr); // run-time error
}

▶ illegal to share an attribute node
```
Refined types for DOM nodes

\[
\text{Node}(\langle di, d, k, f \rangle)
\]

- \(di \ ::= \gamma \mid \text{DOM interface name}\)
  - detecting parent/child mismatches
Refined types for DOM nodes

\[ \text{Node}\langle \text{id}, \text{d}, \text{k}, \text{f} \rangle \]

- \( \text{id} ::= \delta \)
  - owner document
  - detecting owner mismatches
Refined types for DOM nodes

\[ \text{Node}(d_i, d, k, f) \]

- \( k \) ::= \( \kappa \mid A \mid D \)
  
  kinship status (attached A or detached D)

  detecting multiple owners/parents
Refined types for DOM nodes

\[ \text{Node}(d_i, d, k, f) \]

- \( f ::= \phi \mid R \mid F(f) \mid f + f \)
  - kinship degree (abstraction of path to document root)
  - detecting potential cycles
Towards typed attribute ownership

The method **createAttribute**

\[
\forall \delta, \kappa, \phi. \ (\text{Node} \langle \text{Attr}, \delta, D, \phi \rangle) \\
[\text{Node} \langle \text{Document}, \delta, \kappa, R \rangle] \\
\text{createAttribute}(\text{String name})
\]

- abstracting over type modifiers
- return type
- creates detached attribute node
- type of receiver object
  - method of a root document node
- belonging to the receiving document object
Towards typed attribute ownership

The method `setAttributeNode`

∀δ, κ, φ, φ'. (Node⟨Attr, δ, D, φ'⟩)
[Node⟨Element, δ, κ, φ⟩]
setAttributeNode(Node⟨Attr, δ, D, F(φ)⟩)
Towards typed attribute ownership

The method `setAttributeNode`

\[ \forall \delta, \kappa, \phi, \phi'. \ (\text{Node}\langle\text{Attr}, \delta, D, \phi'\rangle) \]
\[ [\text{Node}\langle\text{Element}, \delta, \kappa, \phi\rangle] \]
\[ \text{setAttributeNode} (\text{Node}\langle\text{Attr}, \delta, D, F(\phi)\rangle) \]

▶ takes a detached attribute
Towards typed attribute ownership

The method `setAttributeNode`

\(\forall \delta, \kappa, \phi, \phi'. \ (\text{Node}\langle \text{Attr}, \delta, D, \phi'\rangle)\)

\[\text{setAttributeNode}(\text{Node}\langle \text{Attr}, \delta, D, F(\phi)\rangle)\]

- takes a detached attribute
- attaches it to an element
Towards typed attribute ownership

The method \texttt{setAttributeNode}

\[
\forall \delta, \kappa, \phi, \phi'. \ (\text{Node}\langle \text{Attr}, \delta, D, \phi' \rangle)
\]

\[
[\text{Node}\langle \text{Element}, \delta, \kappa, \phi \rangle]
\]

\texttt{setAttributeNode}(\text{Node}\langle \text{Attr}, \delta, D, F(\phi) \rangle)

- takes a \textbf{detached attribute}
- attaches it to an \textbf{element}
- returns previous (now detached) \textbf{attribute} of same name
Towards typed attribute ownership

The method \texttt{setAttributeNode}

\[
\forall \delta, \kappa, \phi, \phi'. \ (\text{Node}\langle \text{Attr}, \delta, D, \phi' \rangle)
\]

\[
\begin{array}{ll}
\text{setAttributeNode(} & \text{Node}\langle \text{Attr}, \delta, D, F(\phi) \rangle) \\
\end{array}
\]

- takes a detached attribute
- attaches it to an \texttt{element}
- returns previous (now detached) \texttt{attribute} of same name
- \texttt{essential:}
  affine propagation of \texttt{D} property
void highlight (Document doc, Element el1,
    Element el2) {
  Attr attr = doc.createAttribute ("class");
  attr.value = "highlight";
  // attr : Node\langle Attr, d, D, f\rangle
  // split D into D and A
  el1.setAttributeNode (attr);  // needs D property
  // attr : Node\langle Attr, d, A, f\rangle
  el2.setAttributeNode (attr);  // type error
}
More illegal DOM manipulation

Parent-child relations

```java
Document doc = ...
Element el = doc.createElement("center");
```
More illegal DOM manipulation

Parent-child relations

Document doc = ...  
Element el = doc.createElement("center");

▶ Only certain combinations of parent-child nodes are allowed:

Attr attr = doc.createAttribute("class");  
el.appendChild(attr);  // run-time error

Despite the typing Node appendChild (Node), an attribute node cannot become child of an element node
More illegal DOM manipulation
Parent-child relations

Document doc = ...
Element el = doc.createElement("center");

- The underlying graph must remain cycle free:
  el.appendChild (el); // run-time error
Captured by typing of `appendChild`

\[ \forall \delta, \kappa, \phi, \gamma, \gamma'. \ ((\gamma, \gamma') \in \text{PARENTCHILD}) \Rightarrow \]
\[ (\text{Node}(\gamma', \delta, A, F(\phi))) \]
\[ [\text{Node}(\gamma, \delta, \kappa, \phi)] \]
\[ \text{appendChild}(\text{Node}(\gamma', \delta, D, F(\phi))) \]

- Refined type for `appendChild`
Captured by typing of `appendChild`

\[\forall \delta, \kappa, \phi, \gamma, \gamma'. \quad ((\gamma, \gamma') \in \text{PARENTCHILD}) \Rightarrow \]
\[\text{appendChild}(\text{Node}(\gamma', \delta, A, F(\phi)))\]
\[\text{appendChild}(\text{Node}(\gamma, \delta, \kappa, \phi))\]

- **Refined type for `appendChild`**
- **Parent-child relation**
  - abstraction over types \(\gamma\) and \(\gamma'\)
  - pair of types must be in `PARENTCHILD` relation
Captured by typing of `appendChild`

\[ \forall \delta, \kappa, \phi, \gamma, \gamma'. ((\gamma, \gamma') \in \text{PARENTCHILD}) \Rightarrow (\text{Node}(\gamma', \delta, A, F(\phi))) \]

\[ [\text{Node}(\gamma, \delta, \kappa, \phi)] \]

\[ \text{appendChild}(\text{Node}(\gamma', \delta, D, F(\phi))) \]

- Refined type for `appendChild`

- Cycle freedom
  - if parent has kinship degree \( \phi \), then child has kinship degree \( F(\phi) \)
  - object level cycle causes type level cycle \( \phi = F(\phi) \)
  - rejected by occurs check at compile time
Method types

class HiddenAttr {
  Attr anAttr;
  HiddenAttr (Document d, String n, String v) {
    anAttr = d.createAttribute (n);
    anAttr.value = v;
  }
  void attach (Element el) {
    el.setAttributeNode (anAttr);
  }
}

call to attach uses up D property of the attribute
Method types

class HiddenAttr {
    Attr anAttr;
    HiddenAttr (Document d, String n, String v) {
        anAttr = d.createAttribute (n);
        anAttr.value = v;
    }
    void attach (Element el) {
        el.setAttributeNode (anAttr);
    }
}

- call to attach uses up D property of the attribute
- attach should only be called once
class HiddenAttr {
    Attr anAttr;
    HiddenAttr (Document d, String n, String v) {
        anAttr = d.createAttribute (n);
        anAttr.value = v;
    }
    void attach (Element el) {
        el.setAttributeNode (anAttr);
    }
}

► call to attach uses up D property of the attribute
► attach should only be called once
► class type must track kinship state of anAttr
class HiddenAttr {
    Attr anAttr;
    HiddenAttr (Document d, String n, String v) {
        anAttr = d.createAttribute (n);
        anAttr.value = v;
    }
    void attach (Element el) {
        el.setAttributeNode (anAttr);
    }
}

- call to attach uses up D property of the attribute
- attach should only be called once
- class type must track kinship state of anAttr
- method type must check kinship state for fields of this
Type of `attach`

\[ \forall \delta, \kappa, \phi. \text{ void } [A \{ \text{anAttr: Node\{Attr,}\delta, D, F(\phi)\}}]\]

`attach(Node\{Element,}\delta, \kappa, \phi \text{ el})`

Type expresses that:

- D property is used up
- an `Attr` node may be attached to the argument element
- explicit mention of fields requires recursive types for classes, not for annotations
DOMJAVA

- variation of CLASSICJAVA [Flatt, Krishnamurthi, Felleisen]
- additions
  - class types extended with records (and recursion)
  - DOM interface types with annotations
  - method types include the receiver object
  - abstraction over annotations
  - constraints over annotations
- omissions
  - inheritance (has been added)
Kinship Property

Affinity

- kinship $k ::= D \mid A$ is affine property
- implemented by rules for splitting environments and types
- only one use of an D-annotated variable remains D

\[ C \vdash \emptyset \prec \emptyset; \emptyset \]

\[ C \vdash A \prec A_1; A_2 \quad C \vdash t \prec t_1; t_2 \]

\[ C \vdash A, x : t \prec A_1, x : t_1; A_2, x : t_2 \]

\[ C \vdash (k_1 = D \Rightarrow k = D \land k_2 = A) \land (k_2 = D \Rightarrow k = D \land k_1 = A) \]

\[ C \vdash (k = A \iff k_1 = A \land k_2 = A) \]

\[ C \vdash \text{Node} \langle di, d, k, f \rangle \prec \text{Node} \langle di, d, k_1, f \rangle; \text{Node} \langle di, d, k_2, f \rangle \]

\[ (\forall j) \ C \vdash t_j \prec t_j^1; t_j^2 \]

\[ C \vdash c \{ \ldots fd_j : t_j \ldots \} \prec c \{ \ldots fd_j : t_j^1 \ldots \}; c \{ \ldots fd_j : t_j^2 \ldots \} \]
Kinship Property

Subtyping

- an D thing may be used as A, not vice versa
- writing only allowed at declared type (avoids invariance)

\[
\begin{align*}
& C \vdash X \leq X & C \land A \leq B \vdash A \leq B & C \vdash D \leq k \\
& C \vdash f_1 \leq f_2 & C \vdash f \leq f_1 & C \vdash f_1 \leq f \\
& C \vdash F(f_1) \leq F(f_2) & C \vdash f \leq f_1 + f_2 & C \vdash f \leq f_1 + f_2 \\
& C \vdash d_{i_1} \leq d_{i_2} & d_1 = d_2 & C \vdash k_1 \leq k_2 & C \vdash f_1 \leq f_2 \\
& C \vdash \text{Node}(d_{i_1}, d_1, k_1, f_1) \leq \text{Node}(d_{i_2}, d_2, k_2, f_2) \\
& C \vdash c \leq c' \quad (\forall j) & C \vdash t_j \leq t'_j \\
& C \vdash c \{fd_j : t_j\} \leq c' \{fd'_j : t'_j\}
\end{align*}
\]
Technical results

- annotated version of Java’s type system
  - with polymorphism over annotations
  - with annotation subtyping
  - with constraints

- small-step semantics
  - inspired by CLASSICJAVA
  - extended with DOM operations

- type soundness proof that guarantees
  - no shared nodes in DOM graph
  - no cycles in DOM graph
  - no owner mismatches
  - no bad parent-child relationship
Conclusion

- type-based specification on top of Java
- with polymorphic recursion
- based on constrained type system with affine annotations
- extensible to cover almost all DOM runtime errors

Further work

- generalize A/D to other affine properties
  (done: Java(X) @ ECOOP’07)
- improve treatment of container classes
- analysis and implementation (done: Degen’s PhD)
Affine properties

File access

- Permissible operation sequences on file by regular language
- Annotations are regular sublanguages of \((r|w)^*c\)
- Splitting: \(R \prec R_1; R_2\) if \(R_1 \cdot R_2 \subseteq R\)
- Example

```c
/*1*/ File f = fopen("passwd");
>>>>> f : File< (r|w)*c >
>>>>> f’s type is split into File< r > and File< (r|w)*c >

/*2*/ File g = f; // What is g’s type now?
>>>>> g : File< r >; f : File< (r|w)*c >
>>>>> at this point, the system should enforce that all uses of g
>>>>> precede the uses of f, as lined out above

/*3*/ int i = g.read(); // must not use g after f

/*4*/ int j = f.write(); // (r|w)*c >> w | (r|w)*c

/*5*/ int r = f.close(); // (r|w)*c >> c | eps
    // cannot use f or g anymore
```
Coverage of DOM runtime errors

- **HIERARCHY_REQUEST_ERR** is covered except for the case where “the DOM application attempts to append a second DocumentType or Element node to a Document node”. Detecting this error would be possible with a machinery similar to the one detecting nodes that already have parents.

- **WRONG_DOCUMENT_ERR** is covered.

- **NO_MODIFICATION_ALLOWED_ERR** concerns changes to read-only nodes. However, the specification is not quite clear on how read-only nodes may be created and/or recognized in the model. Hence, this property has not be modeled.

- **NOT_SUPPORTED_ERR** deals with removal of nodes from the Document node. This error is not mandatory for all implementations and its treatment would have to be combined with the extended detection of the **HIERARCHY_REQUEST_ERR**.

- **INUSE_ATTRIBUTE_ERR** is covered via parent detection.

- Indexing and bounds check errors are not covered (**DOMSTRING_SIZE_ERR** and **INDEX_SIZE_ERR**).

- **INVALID_CHARACTER_ERR** not covered.

- **NOT_FOUND_ERR** signals that a specific node is not found among the children of a node. This error is not covered because the system would have to retain definite parent information with each child.
forall d₀, f₀.
Node<d₀,O,f₀> nest (Document<d₀,₀,R> d, int n) {
    Node v;
    if (n==0)
        v = d.createTextNode ("The end");
        // rhs : Text<d₀,O,f₀>
        // Text <: Node
        // v : Node<d₀,O,f₀>
    else {
        v = d.createElement ("nest");
        // rhs : Element<d₀,O,f₀>
        // Element <: Node
        // v : Node<d₀,O,f₀>
        v.appendChild (nest (d, n-1));
        // need polymorphic recursion in annotation:
        // nest : Node<d₀,O,F(f₀)>
    }
    return v;
}