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

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

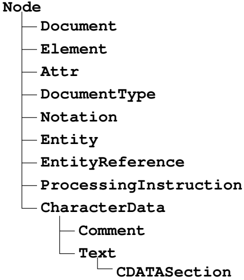
The DOM programming model

- 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

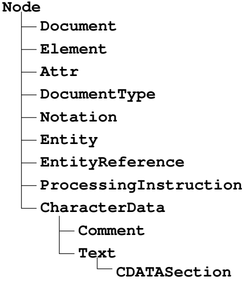


- **Node**
  - **Document**
  - **Element**
  - **Attr**
  - **DocumentType**
  - **Notation**
  - **Entity**
  - **EntityReference**
  - **ProcessingInstruction**
  - **CharacterData**
    - **Comment**
    - **Text**
      - **CDATASection**

Introduction

The DOM programming model

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

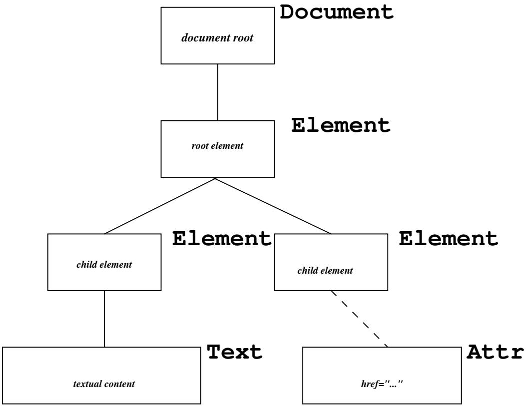


- **`Node`**
  - **`Document`**
  - **`Element`**
  - **`Attr`**
  - **`DocumentType`**
  - **`Notation`**
  - **`Entity`**
  - **`EntityReference`**
  - **`ProcessingInstruction`**
  - **`CharacterData`**
    - **`Comment`**
    - **`Text`**
      - **`CDATASection`**

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

Introduction

What does a typical structure look like?



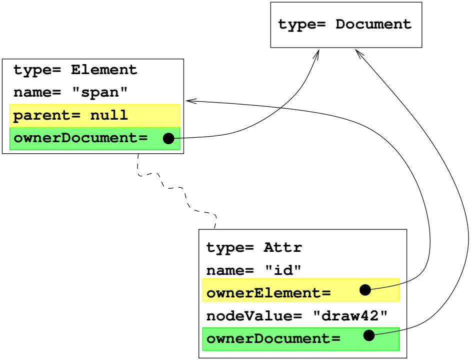
Introduction

What does DOM code look like?

```
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?



type= Document

type= Element
name= "span"
parent= null
ownerDocument=

type= Attr
name= "id"
ownerElement=
nodeValue= "draw42"
ownerDocument=

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

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

```
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

$$\texttt{Node}\langle di, d, k, f\rangle$$

- $di ::= \gamma \mid \text{DOM interface name}$
  detecting parent/child mismatches

Refined types for DOM nodes

$$\texttt{Node}\langle di, d, k, f\rangle$$

- $d ::= \delta$
  owner document
  detecting owner mismatches

Refined types for DOM nodes

$$\texttt{Node}\langle di, d, k, f\rangle$$

- $k ::= \kappa \mid \mathsf{A} \mid \mathsf{D}$
  kinship status (attached A or detached D)
  detecting multiple owners/parents

Refined types for DOM nodes

$$\texttt{Node}\langle di, d, k, f\rangle$$

- $f ::= \phi \mid \mathrm{R} \mid \mathrm{F}(f) \mid f + f$
  kinship degree (abstraction of path to document root)
  detecting potential cycles

Towards typed attribute ownership

The method `createAttribute`

$$\forall \delta, \kappa, \phi. \quad \begin{array}{l} (\texttt{Node}\langle\texttt{Attr}, \delta, \mathsf{D}, \phi\rangle) \\ [\texttt{Node}\langle\texttt{Document}, \delta, \kappa, \mathsf{R}\rangle] \\ \texttt{createAttribute}(\texttt{String}\ \texttt{name}) \end{array}$$

- 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`

$$\begin{array}{ll} \forall \delta, \kappa, \phi, \phi'. & (\texttt{Node}\langle\texttt{Attr}, \delta, \mathsf{D}, \phi'\rangle) \\ & [\texttt{Node}\langle\texttt{Element}, \delta, \kappa, \phi\rangle] \\ & \texttt{setAttributeNode}(\texttt{Node}\langle\texttt{Attr}, \delta, \mathsf{D}, \mathsf{F}(\phi)\rangle) \end{array}$$

Towards typed attribute ownership

The method `setAttributeNode`

$$
\begin{array}{ll}
\forall \delta, \kappa, \phi, \phi'. & (\texttt{Node}\langle \texttt{Attr}, \delta, \mathsf{D}, \phi' \rangle) \\
& [\texttt{Node}\langle \texttt{Element}, \delta, \kappa, \phi \rangle] \\
& \texttt{setAttributeNode}(\texttt{Node}\langle \texttt{Attr}, \delta, \mathsf{D}, \mathsf{F}(\phi) \rangle)
\end{array}
$$

- takes a detached attribute

Towards typed attribute ownership

The method `setAttributeNode`

$$\begin{aligned}\forall \delta, \kappa, \phi, \phi'. \quad & (\texttt{Node}\langle\texttt{Attr}, \delta, \mathsf{D}, \phi'\rangle) \\ & [\texttt{Node}\langle\texttt{Element}, \delta, \kappa, \phi\rangle] \\ & \texttt{setAttributeNode}(\texttt{Node}\langle\texttt{Attr}, \delta, \mathsf{D}, \mathsf{F}(\phi)\rangle)\end{aligned}$$

- takes a detached attribute
- attaches it to an element

Towards typed attribute ownership

The method `setAttributeNode`

$$\begin{aligned}\forall\delta,\kappa,\phi,\phi'.\quad & (\texttt{Node}\langle\texttt{Attr},\delta,\mathsf{D},\phi'\rangle)\\ & [\texttt{Node}\langle\texttt{Element},\delta,\kappa,\phi\rangle]\\ & \texttt{setAttributeNode}(\texttt{Node}\langle\texttt{Attr},\delta,\mathsf{D},\mathsf{F}(\phi)\rangle)\end{aligned}$$

- takes a detached attribute
- attaches it to an element
- returns previous (now detached) attribute of same name

Towards typed attribute ownership

The method `setAttributeNode`

$$\forall \delta, \kappa, \phi, \phi'. \quad \begin{array}{l} (\mathtt{Node}\langle \mathtt{Attr}, \delta, \mathsf{D}, \phi' \rangle) \\ [\mathtt{Node}\langle \mathtt{Element}, \delta, \kappa, \phi \rangle] \\ \mathtt{setAttributeNode}(\mathtt{Node}\langle \mathtt{Attr}, \delta, \mathsf{D}, \mathsf{F}(\phi) \rangle) \end{array}$$

- takes a detached attribute
- attaches it to an element
- returns previous (now detached) attribute of same name
- essential:
  affine propagation of D property

Towards typed attribute ownership

Untypable DOM manipulation

```
void highlight (Document doc, Element el1,
                              Element el2) {
  Attr attr = doc.createAttribute ("class");
  attr.value = "highlight";
  // attr : Node⟨Attr, d, D, f⟩
  // split D into D and A
  el1.setAttributeNode (attr);  // needs D property
  // attr : Node⟨Attr, d, A, f⟩
  el2.setAttributeNode (attr);  // type error
}
```

More illegal DOM manipulation

Parent-child relations

```
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`

$$\begin{array}{ll} \forall \delta, \kappa, \phi, \gamma, \gamma'. & ((\gamma, \gamma') \in \textsf{P}\textsf{\scriptsize ARENT}\textsf{C}\textsf{\scriptsize HILD}) \Rightarrow \\ & (\texttt{Node}\langle \gamma', \delta, \mathsf{A}, \mathsf{F}(\phi) \rangle) \\ & [\texttt{Node}\langle \gamma, \delta, \kappa, \phi \rangle] \\ & \texttt{appendChild}(\texttt{Node}\langle \gamma', \delta, \mathsf{D}, \mathsf{F}(\phi) \rangle) \end{array}$$

- Refined type for `appendChild`

Captured by typing of `appendChild`

$$\forall \delta, \kappa, \phi, \gamma, \gamma'. \quad \begin{array}{l} ((\gamma, \gamma') \in \textsc{ParentChild}) \Rightarrow \\ (\texttt{Node}\langle \gamma', \delta, \mathsf{A}, \mathsf{F}(\phi) \rangle) \\ [\texttt{Node}\langle \gamma, \delta, \kappa, \phi \rangle] \\ \texttt{appendChild}(\texttt{Node}\langle \gamma', \delta, \mathsf{D}, \mathsf{F}(\phi) \rangle) \end{array}$$

- Refined type for `appendChild`
- Parent-child relation
    - abstraction over types γ and γ'
    - pair of types must be in ParentChild relation

Captured by typing of `appendChild`

$$\begin{array}{ll}\forall \delta, \kappa, \phi, \gamma, \gamma'. & ((\gamma, \gamma') \in \textsc{ParentChild}) \Rightarrow \\ & (\texttt{Node}\langle \gamma', \delta, \mathsf{A}, \mathsf{F}(\phi)\rangle) \\ & [\texttt{Node}\langle \gamma, \delta, \kappa, \phi\rangle] \\ & \texttt{appendChild}(\texttt{Node}\langle \gamma', \delta, \mathsf{D}, \mathsf{F}(\phi)\rangle)\end{array}$$

- Refined type for `appendChild`

- Cylcle freedom
  - if parent has kinship degree ϕ, then child has kinship degree $\mathsf{F}(\phi)$
  - object level cycle causes type level cycle $\phi = \mathsf{F}(\phi)$
  - 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

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 type must track kinship state of `anAttr`

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 type must track kinship state of `anAttr`
- method type must check kinship state for fields of `this`

Type of `attach`

$$\begin{array}{ll} \forall \delta, \kappa, \phi. & \texttt{void} \\ & [\texttt{A}\ \{\texttt{anAttr} : \texttt{Node}\langle \texttt{Attr}, \delta, \mathsf{D}, \mathsf{F}(\phi)\rangle\}] \\ & \texttt{attach}(\texttt{Node}\langle \texttt{Element}, \delta, \kappa, \phi\rangle\ \texttt{el}) \end{array}$$

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 | A is *affine property*
- implemented by rules for splitting environments and types
- only one use of an D-annotated variable remains D

$$C \vdash \emptyset \prec \emptyset ; \emptyset \qquad \frac{C \vdash A \prec A_1 ; A_2 \quad C \vdash t \prec t_1 ; t_2}{C \vdash A, x : t \prec A_1, x : t_1 ; A_2, x : t_2}$$

$$\frac{\begin{array}{c} C \Vdash (k_1 = \mathrm{D} \Rightarrow k = \mathrm{D} \wedge k_2 = \mathrm{A}) \wedge (k_2 = \mathrm{D} \Rightarrow k = \mathrm{D} \wedge k_1 = \mathrm{A}) \\ C \Vdash (k = \mathrm{A} \Leftrightarrow k_1 = \mathrm{A} \wedge k_2 = \mathrm{A}) \end{array}}{C \vdash \mathtt{Node}\langle di, d, k, f \rangle \prec \mathtt{Node}\langle di, d, k_1, f \rangle ; \mathtt{Node}\langle di, d, k_2, f \rangle}$$

$$\frac{(\forall j)\ C \vdash t_j \prec t_j^1 ; t_j^2}{C \vdash c\ \{ \dots fd_j : t_j \dots \} \prec c\ \{ \dots fd_j : t_j^1 \dots \} ; c\ \{ \dots fd_j : t_j^2 \dots \}}$$

Kinship Property

Subtyping

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

$$C \vdash X \leq X \qquad C \wedge A \leq B \vdash A \leq B \qquad C \vdash \mathrm{D} \leq k$$

$$\frac{C \vdash f_1 \leq f_2}{C \vdash \mathrm{F}(f_1) \leq \mathrm{F}(f_2)} \qquad \frac{C \vdash f \leq f_1}{C \vdash f \leq f_1 + f_2} \qquad \frac{C \vdash f_1 \leq f \quad C \vdash f_2 \leq f}{C \vdash f_1 + f_2 \leq f}$$

$$\frac{C \vdash di_1 \leq di_2 \quad d_1 = d_2 \quad C \vdash k_1 \leq k_2 \quad C \vdash f_1 \leq f_2}{C \vdash \mathtt{Node}\langle di_1, d_1, k_1, f_1\rangle \leq \mathtt{Node}\langle di_2, d_2, k_2, f_2\rangle}$$

$$\frac{C \vdash c \leq c' \quad (\forall j)\ C \vdash t_j \leq t'_j}{C \vdash c\ \{fd_j : t_j\} \leq c'\ \{fd'_j : t'_j\}}$$

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

```
/⋆1⋆/ File f = fopen("passwd");
>>>>>   f : File< (r|w)⋆c >
>>>>>   f’s type is split into File< r > and Fil
/⋆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
>>>>>   precede the uses of f, as lined out abov
/⋆3⋆/ int i = g.read(); // must not use g after
/⋆4⋆/ int j = f.write();   // (r|w)⋆c >> w | (r|
/⋆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.

Recursion

```
forall d0, f0.
Node<d0,O,f0> nest (Document<d0,O,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;
}
```