Soutenance de thèse de doctorat de

Benjamin Canou

sous la direction d'Emmanuel Chailloux et Vincent Balat

Programmation Web Typée

Paris, le 4 octobre 2011

Université Pierre et Marie Curie

École Doctorale Informatique, Télécommunications et Électronique

Programmation Web Typée ?

Vous avez dit Programmation Web Typée ?

L'aspect étudié

Programmer c'est

- concevoir des algorithmes ;
- les implanter dans un langage de programmation.

L'aspect étudié

Programmer c'est

- concevoir des algorithmes ;
- les implanter dans un langage de programmation.

Ce qui nous intéresse : les langages et leur conception.

1. La syntaxe (la grammaire).
2. Les paradigmes (fonctionnel, impératif, etc.).
3. Les modèles de concurrence (préemptif, coopératif, etc.).
4. Les systèmes de types.

Caractéristique importante d'un langage

Typer un programme c'est :

- assigner un type aux données (entier, texte, fruit, orange) ;
- n'appliquer les calculs que sur des données du type prévu.

Plusieurs grandes familles de systèmes de types :

- typage dynamique ;
- typage statique, celle qui nous intéresse.

Le sujet d'étude

Qu'est-ce que le Web ? Essentiellement :

1. Un format de document (hyper)textuels. (HTML)
2. Des (hyper)liens entre ces documents. (URL)
3. Un protocole réseau non connecté. (HTTP)

Le sujet d'étude

Qu'est-ce que le Web ? Essentiellement :

1. Un format de document (hyper)textuels. (HTML)
2. Des (hyper)liens entre ces documents. (URL)
3. Un protocole réseau non connecté. (HTTP)

Et le Web moderne :

1. Mises-à-jour du document côté client. (JavaScript/DOM)
2. Communications plus complexes, agrégation. (AJAX)
3. Multi-média. (HTML5)

Recette pour une application Web

1. Pour la partie serveur :
   - génération de documents ;
   - accès aux bases de données ;
   - gestion d'utilisateurs ;
   - code métier.
2. Pour la partie navigateur :
   - formatage visuel ;
   - inter-actions avec l'utilisateur ;
   - modifications du document ;
3. Et pour les communications :
   - gestion de connexions ;
   - mises-à-jour partielles.

Recette pour une application Web

5+ langages

1. Pour la partie serveur :
   - génération de documents ; — dédié (ex. PHP) ou généraliste
   - accès aux bases de données ; — SQL
   - gestion d'utilisateurs ;
   - code métier.
2. Pour la partie navigateur :
   - formatage visuel ; — HTML, CSS
   - inter-actions avec l'utilisateur ; — JavaScript
   - modifications du document ;
3. Et pour les communications :
   - gestion de connexions ;
   - mises-à-jour partielles.

Recette pour une application Web

5+ langages, 5+ formats de données

1. Pour la partie serveur :
   - génération de documents ; — données du langage
   - accès aux bases de données ; — données de la base
   - gestion d'utilisateurs ;
   - code métier.
2. Pour la partie navigateur :
   - formatage visuel ; — valeurs CSS
   - inter-actions avec l'utilisateur ; — données JavaScript
   - modifications du document ;
3. Et pour les communications :
   - gestion de connexions ;
   - mises-à-jour partielles. — sérialisation

Recette pour une application Web

5+ langages, 5+ formats de données, 3 modèles de document

1. Pour la partie serveur :
   - génération de documents ; modèle abstrait
   - accès aux bases de données ;
   - gestion d'utilisateurs ;
   - code métier.
2. Pour la partie navigateur :
   - formatage visuel ;
   - inter-actions avec l'utilisateur ;
   - modifications du document ; DOM
3. Et pour les communications :
   - gestion de connexions ;
   - mises-à-jour partielles. XML

Recette pour une application Web

5+ langages, 5+ formats de données, 3 modèles de document

...et autant de sources de failles et de bogues.

1. Pour la partie serveur :
   - génération de documents ;
   - accès aux bases de données ;
   - gestion d'utilisateurs ;
   - code métier.
2. Pour la partie navigateur :
   - formatage visuel ;
   - inter-actions avec l'utilisateur ;
   - modifications du document ;
3. Et pour les communications :
   - gestion de connexions ;
   - mises-à-jour partielles.

On cherche à programmer le Web avec :
Un langage, un format de données, un modèle de document

Point de départ : Ocsigen, programmation Web en OCaml.

1. OCaml pour expérimenter :
   - langage multi-paradigmes (fonctionnel, impératif et autres) ;
   - modèle de concurrence libre ;
   - quel est le bon modèle pour programmer le Web ?
2. OCaml pour la sûreté :
   - sûreté d'exécution par typage statique fort ;
   - génération de pages HTML valides ;
   - typage statique de l'interaction ;
   - typage des requêtes aux bases de données.

On cherche à programmer le Web avec :

Un langage, un format de données, un modèle de document

Contributions :

1. Programmation du navigateur en OCaml.
   - un seul langage, un seul système de types ;
   - expérimentations de modèles côté client.
2. Modèle sûr et uniforme de document.
   - Un seul modèle pour toutes les parties ;
   - création et manipulation ;
   - possibilité de typage statique.

Première partie

Programmation du client

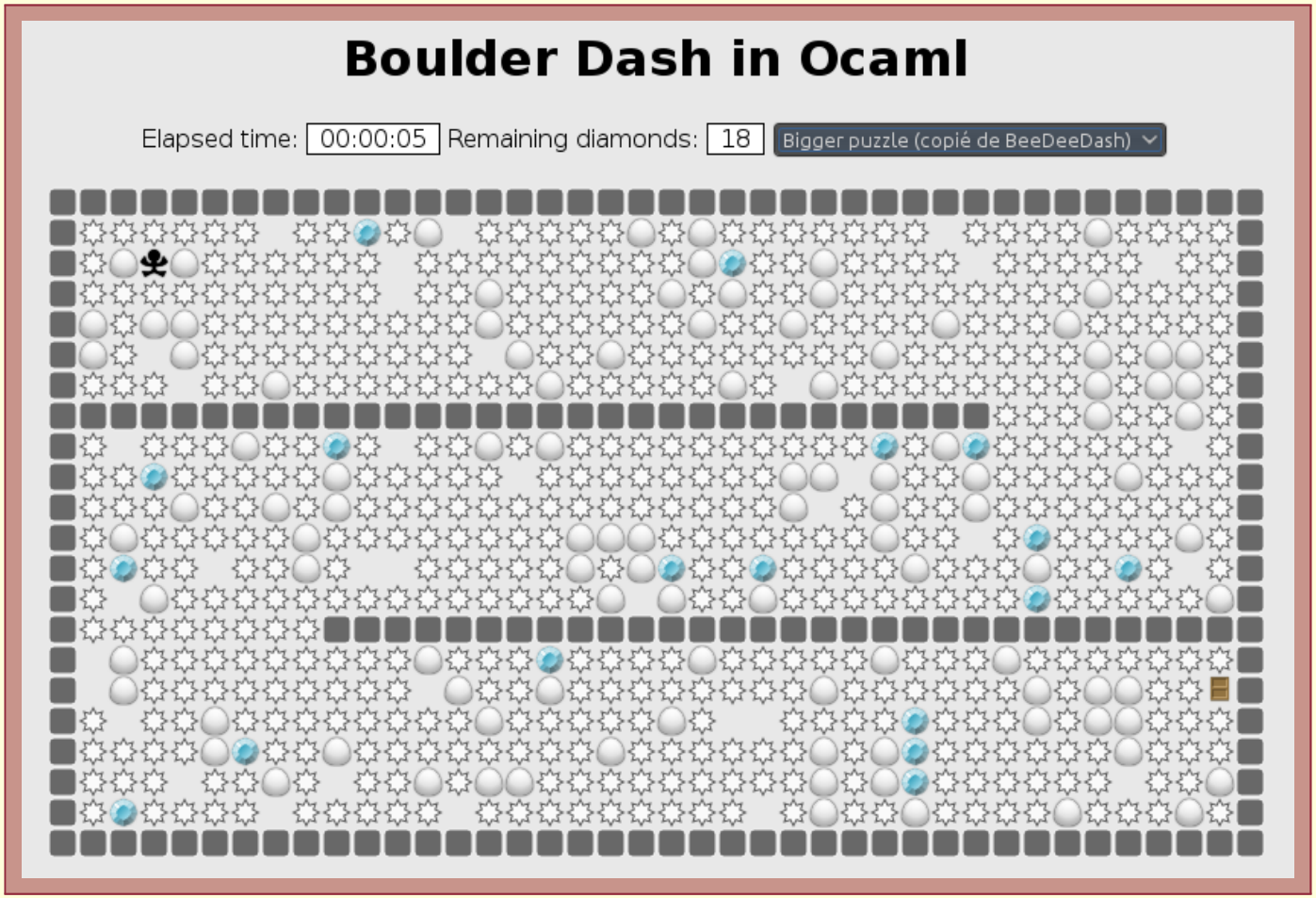
L'expérience OBrowser

Pour utiliser un langage dans le navigateur :

1. Greffons (plug-ins) :
   - \+ performances potentiellement bonnes ;
   - \- interactions limitées avec la page ;
   - \- nécessite une installation.
2. Compilation vers JavaScript :
   - \- performances de JavaScript ;
   - \- difficulté de mise au point ;
   - \+ interaction complète avec la page.

Pour utiliser un langage dans le navigateur :

1. Greffons (plug-ins) :
   + performances potentiellement bonnes ;
   - interactions limitées avec la page ;
   - nécessite une installation.
2. Compilation vers JavaScript :
   - performances de JavaScript ;
   - difficulté de mise au point ;
   + interaction complète avec la page.
3. OBrowser : une machine virtuelle OCaml en JavaScript :
   - performances limitées attendues ;
   + maintenance plus facile qu'un compilateur ;
   + sémantique identique, performances prévisibles ;
   + format des données compatible avec le serveur ;
   + gain en abstraction (ex. concurrence préemptive).



- Performances trop limitées pour des programmes intensifs ;
- suffisantes pour programmer l'interaction (ex. petits jeux) ;
- ex : Boulder Dash, rendu graphique par le DOM.

1. OCaml sur les deux parties :
   - Données compatibles, transmission de valeurs OCaml ;
   - prise en charge de tout le langage ;
   - mise à disposition du DOM en OCaml ;
   - des performances suffisantes pour les besoins sur le client ;
   - $\Rightarrow$ OBrowser inclus dans Ocsigen 1.3 ;
   - sûreté par typage statique de l'application entière.

1. OCaml sur les deux parties :
   - Données compatibles, transmission de valeurs OCaml ;
   - prise en charge de tout le langage ;
   - mise à disposition du DOM en OCaml ;
   - des performances suffisantes pour les besoins sur le client ;
   - $\Rightarrow$ OBrowser inclus dans Ocsigen 1.3 ;
   - sûreté par typage statique de l'application entière.
2. Mais encore trois modèles de document !

Seconde partie

Un nouveau modèle de document

La problématique : pourquoi pas DOM ?

Deux listes (rendu du navigateur)

- Item A
- Item B

- Item X
- Item Y

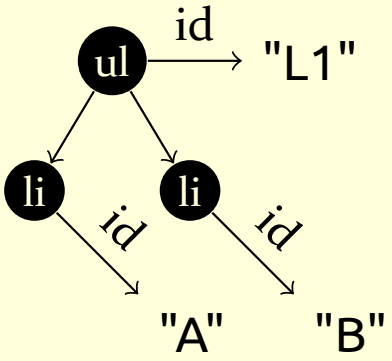
Deux listes (code source)

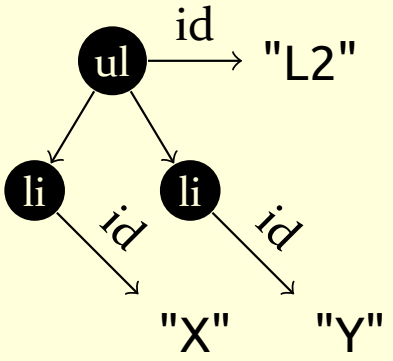
```
1: <ul id="L1">
2:   <li id="A">Item A</li>
3:   <li id="B">Item B</li>
4: </ul>
```

```
1: <ul id="L2">
2:   <li id="X">Item X</li>
3:   <li id="Y">Item Y</li>
4: </ul>
```

- document valide (bien typé par rapport à la grammaire HTML)
  - un ul ne contient que des li ;
  - chaque ul contient au moins un li ;
- validité vérifiable automatiquement ;
- validité de la génération statiquement typable (CDuce, etc.) ;
- seule difficulté : les ids.

Deux listes (DOM)

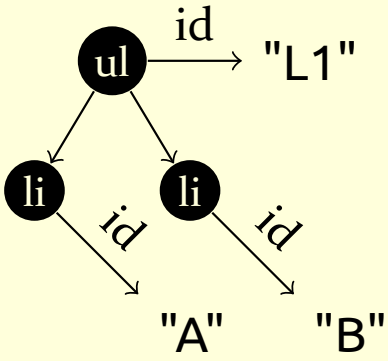


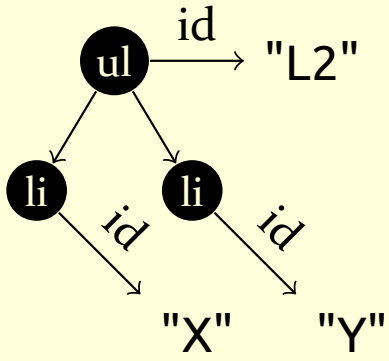


Deux listes (DOM)

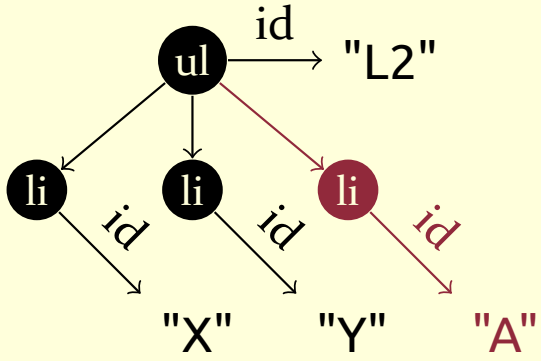

```
1: let a = Dom.get_element_by_id "A"
2: and l2 = Dom.get_element_by_id "L2" in
3:   Dom.append_child l2 a
```

Deux listes (DOM)

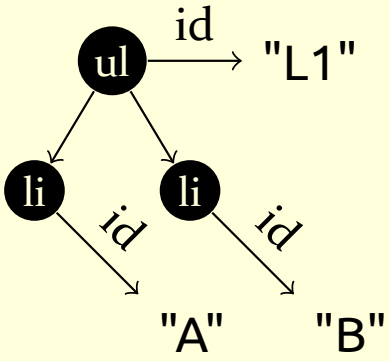


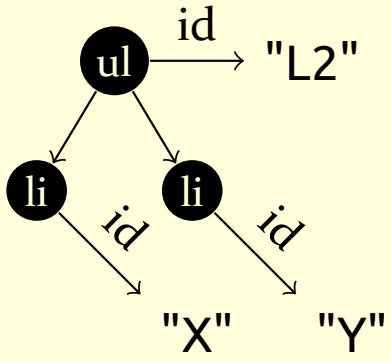



```
1: let a = Dom.get_element_by_id "A"
2: and l2 = Dom.get_element_by_id "L2" in
3:   Dom.append_child l2 a
```

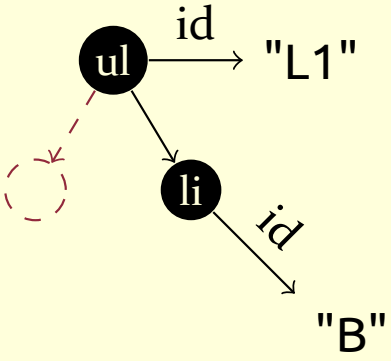


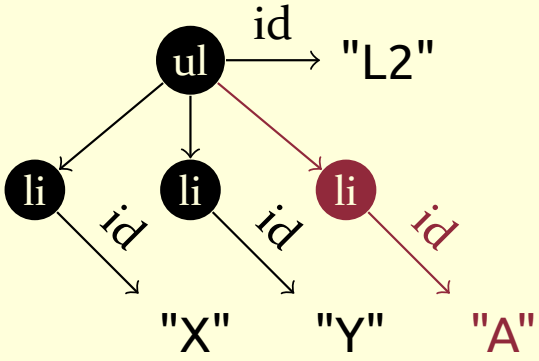
Deux listes (DOM)






```
1: let a = Dom.get_element_by_id "A"
2: and l2 = Dom.get_element_by_id "L2" in
3:   Dom.append_child l2 a
```





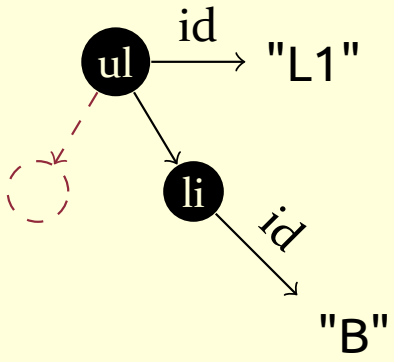
Pour la cohérence de l'état du moteur de rendu, les objets du DOM doivent former un arbre :

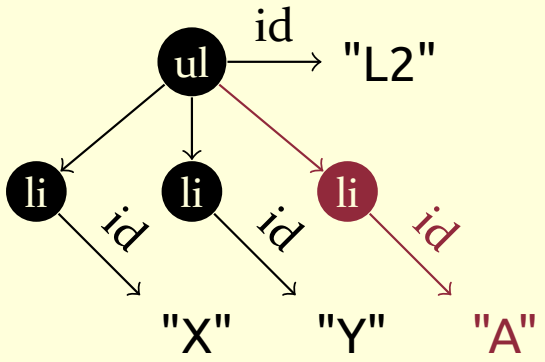
- Pas de cycle ;
  (pas d'affichage récursif infini).
- pas de partage.
  (élément affiché à un seul endroit dans la page).

Mécanisme

- logique si on connaît l'implantation ;
- mais source d'imprévu.
  (difficile de maîtriser le comportement)

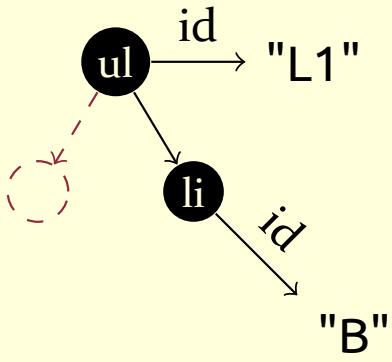
Introduction d'une erreur de typage

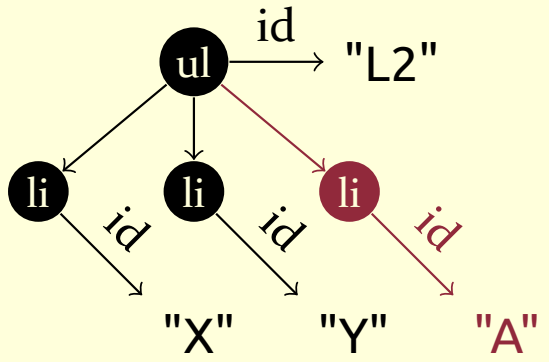




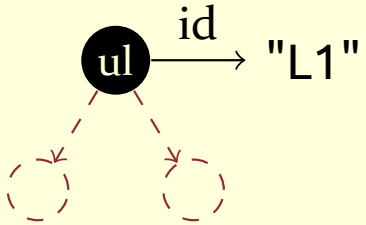

```
1: let b = Dom.get_element_by_id "B"
2: and l2 = Dom.get_element_by_id "L2" in
3:   Dom.append_child l2 b
```

Introduction d'une erreur de typage

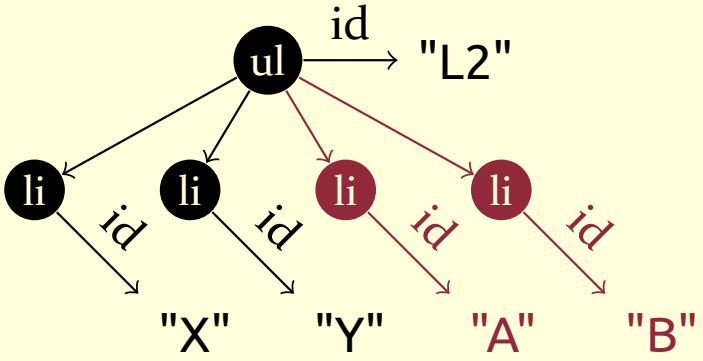





```
1: let b = Dom.get_element_by_id "B"
2: and l2 = Dom.get_element_by_id "L2" in
3:   Dom.append_child l2 b
```



Mal typée !



Exemple de création de nœuds :

```
1: let a = li [ cdata "Item A" ]
2: let l1 = ul [ a ]
3: let l2 = ul [ a ]
```

Déroulement :

Exemple de création de nœuds :

```
1: let a = li [ cdata "Item A" ]
2: let l1 = ul [ a ]
3: let l2 = ul [ a ]
```

Déroulement :

1. On crée a de façon bien typée.
2. On crée l1 de façon bien typée.

Exemple de création de nœuds :

```
1: let a = li [ cdata "Item A" ]
2: let l1 = ul [ a ]
3: let l2 = ul [ a ]
```

Déroulement :

1. On crée a de façon bien typée.
2. On crée l1 de façon bien typée.
3. On crée l2 de façon bien typée

Exemple de création de nœuds :

```
1: let a = li [ cdata "Item A" ]
2: let l1 = ul [ a ]
3: let l2 = ul [ a ]
```

Déroulement :

1. On crée a de façon bien typée.
2. On crée l1 de façon bien typée.
3. On crée l2 de façon bien typée mais a est supprimé de l1 !
4. la création de l2 rend l1 mal typée.

Conclusion :

- Les manipulations cassent le typage de valeurs déjà construites.
- La construction casse le typage de valeurs déjà construites.

1. Solution de Ocsigen, HOP :
   - création : typée (pour Ocsigen) et sans vérification de partage ;
   - manipulations : API non typée pour les effets..
2. Solution de Links, OPA :
   - création : représentation intermédiaire ;
   - manipulations : représentation intermédiaire pour les affectés.
3. Travaux de Peter Thiemann :
   - création : refus via le système de types ;
   - manipulations : refus via le système de types.

1. Solution de Ocsigen, HOP :
   - création : typée (pour Ocsigen) et sans vérification de partage ;
   - manipulations : API non typée pour les effets..
2. Solution de Links, OPA :
   - création : représentation intermédiaire ;
   - manipulations : représentation intermédiaire pour les affectés.
3. Travaux de Peter Thiemann :
   - création : refus via le système de types ;
   - manipulations : refus via le système de types.
4. Notre solution :
   - création : copies implicites ;
   - manipulations : copies implicites.

Que copier ?

Une page Web, ça n'est pas qu'un arbre :

- copy doit dupliquer les données associées ;

```
1: <a style={ color: red ; background: yellow }>Click !</a>
```

- ainsi que les fermetures.

```
1: <a onclick=(fun () -> alert "hello !")>Click !</a>
```

Où s'arrêter ?

- Partage ou copie pour cpt ?

```
1: let cpt = ref 0 in
2:   <a onclick=(fun () -> incr cpt)>Click !</a>
```

- Et que faire pour les cycles ?

```
1: let cpt = ref 0 in
2: let rec self =
3:   <a onclick=(fun () ->
4:              incr cpt ;
5:              append self (sprintf "%d clicks" !cpt))>Click !</a>
```

1. Construction syntaxique délimitée pour les nœuds.
2. On duplique ce qui est dans la portée.

Compteur partagé :

```
1: let cpt = ref 0 in
2:  node <a>
3:   [ "Click" ]
4:   prop onClick = fun () ->
5:    incr cpt
6: end
```

Compteur dupliqué :

```
1: node <a>
2:  let cpt = ref 0 in
3:   [ "Click" ]
4:   prop click = fun () ->
5:    incr cpt
6: end
```

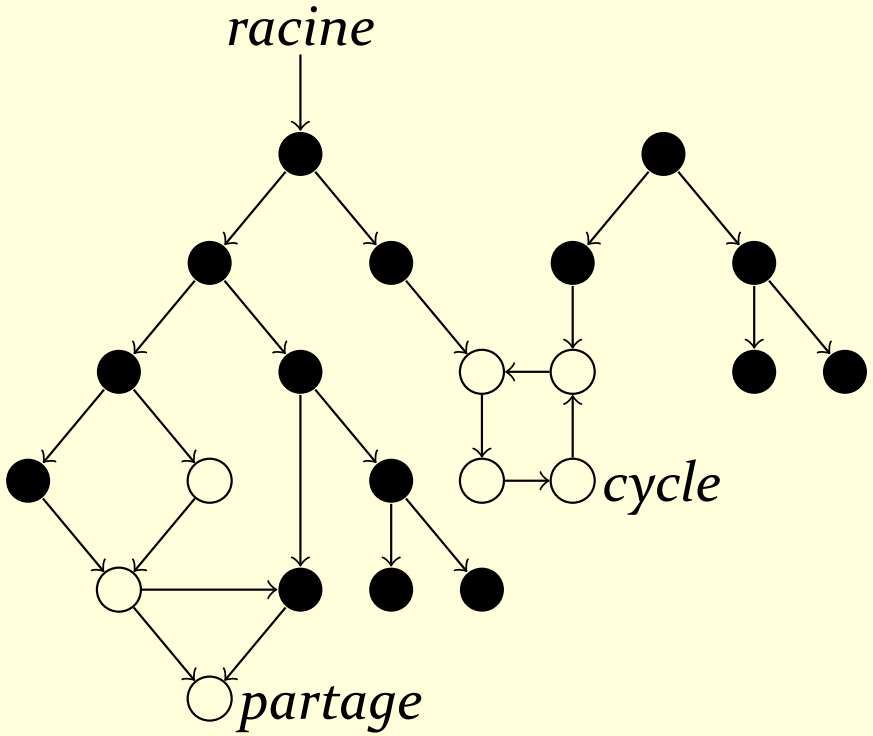
Un nouveau modèle de document

Formalisation

1. Une formalisation du DOM actuel : ^{f}DOM .
2. Une version alternative avec copies : ^{c}DOM .
3. Un langage de manipulation bien typé : FidoML .

On voit le document comme un graphe bicolore :

- document proprement dit : arbre des nœuds noirs (●) ;
- données annexes : graphe des nœuds blancs (○).



- Ensemble de primitives (API, à la manière du DOM).
- Règles décrivant les effets de ces primitives sur l'état.
- Définition formelle de l'état du document : (H, L, T, P)
  - H (Heap) $= H^{\bullet} \cup H^{\circ}$: nœuds et données annexes,
  - L (Labels) : étiquettes des nœuds,
  - T (Tree) : structure d'arbre,
  - P (Properties) : arcs étiquetés du graphe.

- Règles d'accès : $\mathsf{tag} : H^{\bullet} \rightarrow Tag$

$$(\textsc{Tag})\ \frac{(\bullet, t) \in L}{(H, L, T, P) \vdash \mathsf{tag}(\bullet) = t, (H, L, T, P)}$$

- Règles d'accès : tag : $H^{\bullet} \rightarrow Tag$
- Règles d'effets : set : $H \times Key \times H \cup Imm \rightarrow \{nil\}$

$$(\textsc{Set})\ \frac{\nexists v', (\bullet, k, v') \in P}{(H, L, T, P) \vdash \mathsf{set}(\bullet, k, v) = nil, (H, L, T, P \cup (\bullet, k, v))}$$

$$(\textsc{Modify})\ \frac{\exists v' (\bullet, k, v') \in P}{(H, L, T, P) \vdash \mathsf{set}(\bullet, k, v) = nil, (H, L, T, P \backslash \{(\bullet, k, v')\} \cup (\bullet, k, v))}$$

- Règles d'accès : tag : $H^\bullet \rightarrow Tag$
- Règles d'effets : set : $H \times Key \times H \cup Imm \rightarrow \{nil\}$
- Déplacements implicites : bind : $H^\bullet \times H^\bullet \rightarrow \{nil\}$

$$(\textsc{Attach})\ \frac{\begin{array}{c}\bullet_p \in H^\bullet \quad \bullet_n \in H^\bullet \\ \bullet_{p'} \in H^\bullet \Rightarrow \bullet_n \notin T(\bullet_{p'}) \\ \neg chain(T, \bullet_n, \bullet_p)\end{array}}{(H, L, T, P) \vdash \mathsf{bind}(\bullet_p, \bullet_n) = nil, (H, L, T', P)}$$

$$\text{où } T' = T \backslash \{\bullet_p, T(\bullet_p)\} \cup \{\bullet_p, \bullet_n :: T(\bullet_p)\}$$

$$(\textsc{Move})\ \frac{\begin{array}{c}\bullet_p \in H^\bullet \quad \bullet_n \in H^\bullet \\ \exists \bullet_{p'}, \bullet_n \in T(\bullet_{p'}) \\ \neg chain(T, \bullet_n, \bullet_p)\end{array}}{(H, L, T, P) \vdash \mathsf{bind}(\bullet_p, \bullet_n) = nil, (H, L, T', P)}$$

$$\text{où } T' = T \backslash \{(\bullet'_p, T(\bullet'_p)), (\bullet_p, T(\bullet_p))\} \cup \{(\bullet'_p, T(\bullet'_p) - \bullet_n), (\bullet_p, \bullet_n :: T(\bullet_p))\}$$

Rappel du principe :

- On copie implicitement au lieu de déplacer.
- On utilise la portée du nœud pour déterminer quoi copier.

Extension de ^{f}DOM:

- (H, L, T, P, S, s)
  - S (Scoping) : informations de portée,
  - s (Stack) : pile des portées ouvertes.
- Nouvelles primitives.
- Mise-à-jour des règles.

Primitive	Type	Règles
children	$H^{\bullet} \rightarrow Int$	Children
child	$H^{\bullet} \times Int \rightarrow H^{\bullet} \cup \{nil\}$	Child, Unbound
roots	$\{nil\} \rightarrow Enum(H^{\bullet})$	Roots
properties	$H \rightarrow Enum(Key)$	Properties
get	$H \times Key \rightarrow H \cup Prim \cup \{nil\}$	Get, Get-Unbound
tag	$H \rightarrow Tag$	Tag
create$^{\bullet}$	$Tag \rightarrow H^{\bullet}$	Create$^{\bullet}$
create$^{\circ}$	$\{nil\} \rightarrow H^{\circ}$	Create$^{\circ}$
close	$\{nil\} \rightarrow \{nil\}$	Close-Scope
reopen	$H^{\bullet} \rightarrow \{nil\}$	Reopen-Scope
detach	$H^{\bullet} \rightarrow \{nil\}$	Detach-1, Detach-2
copy	$H^{\bullet} \times H^{\bullet} \rightarrow H^{\bullet}$	Copy
bind	$H^{\bullet} \times H^{\bullet} \rightarrow \{nil\}$	Attach, Attach-Copy
set	$H \times Key \times H \cup Prim \rightarrow \{nil\}$	Set, Modify
unset	$H \times Key \rightarrow \{nil\}$	Set, Modify

Gestion de la portée

Trois opérations effectuées :

- ouverture ;
  - à la création d'un nœud noir : `create`$^{\bullet}$;
  - à la réouverture explicite : `reopen` ;
  - empile le nœud dans s ;

Gestion de la portée

Trois opérations effectuées :

- ouverture ;
  - à la création d'un nœud noir : create$^{\bullet}$;
  - à la réouverture explicite : reopen ;
  - empile le nœud dans s ;
- mise-à-jour des informations ;
  - à l'allocation : create$^{\bullet}$, create$^{\circ}$;
  - associe le nœud alloué au nœud dont la portée est ouverte ;

Gestion de la portée

Trois opérations effectuées :

- ouverture ;
  - à la création d'un nœud noir : create$^{\bullet}$;
  - à la réouverture explicite : reopen ;
  - empile le nœud dans s ;
- mise-à-jour des informations ;
  - à l'allocation : create$^{\bullet}$, create$^{\circ}$;
  - associe le nœud alloué au nœud dont la portée est ouverte ;
- fermeture.
  - explicitement avec : close ;
  - dépile un nœud dans s.

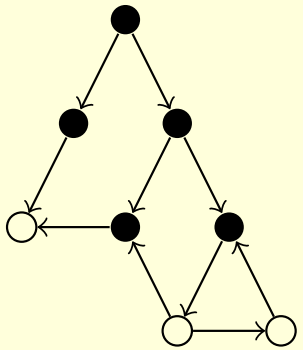
Copie explicite

$$(\textsc{Copy})\ \frac{\ldots}{(H, L, T, P, S, s) \ \vdash\ \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

Copie explicite

$$(\textsc{Copy})\ \frac{\cdots}{(H, L, T, P, S, s) \ \vdash\ \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

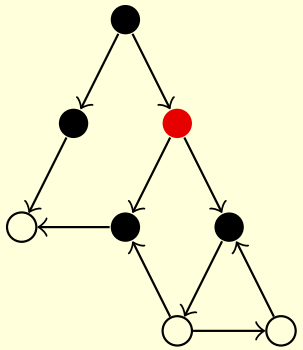
1. Graphe initial.



Copie explicite

$$(\textsc{Copy})\ \frac{\ldots}{(H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

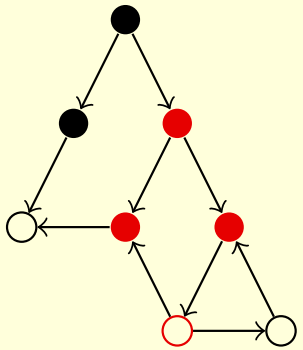
1. Graphe initial.
2. Nœud à copier.



Copie explicite

$$(\textsc{Copy})\ \frac{\ldots}{(H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

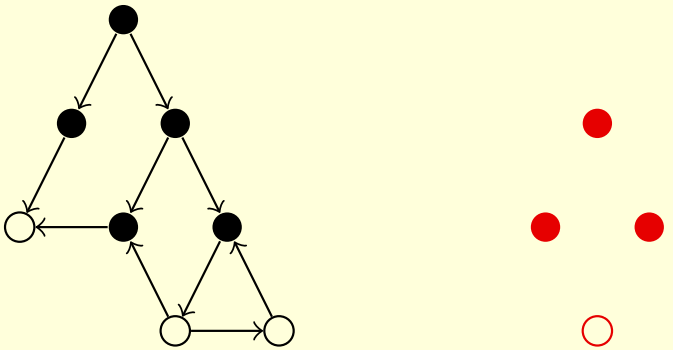
1. Graphe initial.
2. Nœud à copier.
3. Nœuds dans la portée.



Copie explicite

$$(\textsc{Copy})\ \frac{\cdots}{(H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

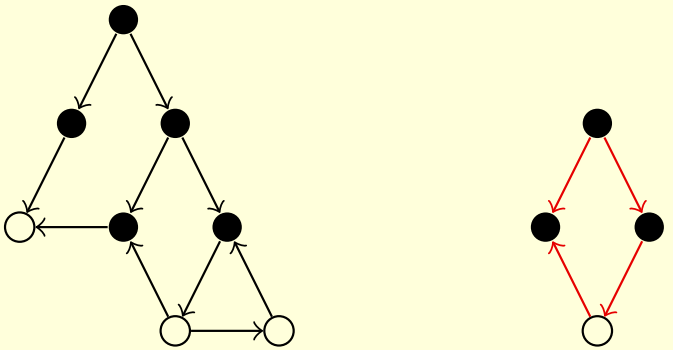
1. Graphe initial.
2. Nœud à copier.
3. Nœuds dans la portée.
4. Duplication.



Copie explicite

$$(\textsc{Copy})\ \frac{\ldots}{(H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

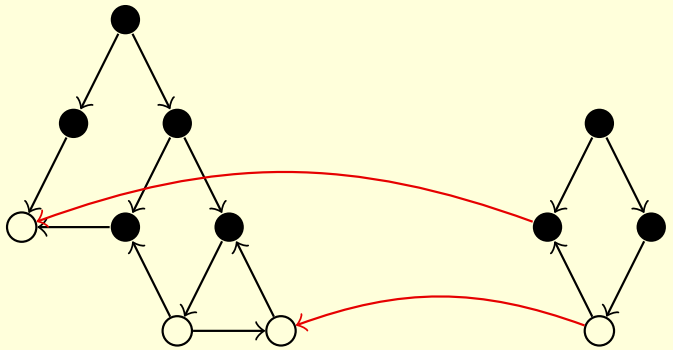
1. Graphe initial.
2. Nœud à copier.
3. Nœuds dans la portée.
4. Duplication.
5. Reliaison interne.



Copie explicite

$$(\textsc{Copy})\ \frac{\ldots}{(H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

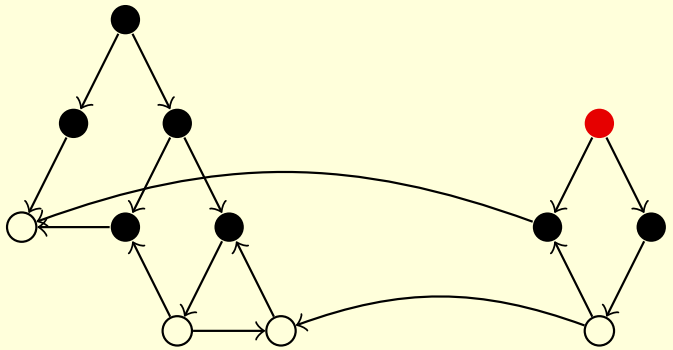
1. Graphe initial.
2. Nœud à copier.
3. Nœuds dans la portée.
4. Duplication.
5. Reliaison interne.
6. Reliaison externe.



Copie explicite

$$(\textsc{Copy})\ \frac{\cdots}{(H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', L', T', P', S', s)}$$

1. Graphe initial.
2. Nœud à copier.
3. Nœuds dans la portée.
4. Duplication.
5. Reliaison interne.
6. Reliaison externe.
7. Résultat de la copie.



Copie implicite

La règle (Attach-Copy) remplace (Move).

$$(\textsc{Attach-Copy})\ \frac{\begin{array}{c}\bullet_p \in H^\bullet \quad \bullet_n \in H^\bullet \\ \exists \bullet_{p'}, \bullet_n \in T(\bullet_{p'}) \vee \mathit{chain}(T, \bullet_n, \bullet_p) \\ (H, L, T, P, S, s) \vdash \mathsf{copy}(\bullet_n) = \bullet_{n'}, (H', T', P', S', s)\end{array}}{(H, L, T, P, S, s) \vdash \mathsf{bind}(\bullet_p, \bullet_n) = \mathit{nil}, (H', L', T'', P', S', s)}$$

où $T'' = T' \backslash \{\bullet_p, T'(\bullet_p)\} \cup \{\bullet_p, \bullet_{n'} :: T'(\bullet_p)\}$

Un nouveau modèle de document

FidoML : un langage pour manipuler le document

Syntaxe à la Caml classique, étendue avec :

- Création de nœuds :

  rec-expr += node <*tag*> *expr* [prop *id* = *expr*]* end

  - étiquette ;
  - propriétés valuées ;
  - enfants du nœuds, sous forme de liste.

- Opérations sur les enfants :

  expr += children *expr* | replace *expr expr*

  - children donne la séquence des enfants sous forme de liste ;
  - replace remplace les enfants d'un nœud par une liste.

- Opérations sur les propriétés :

  expr += *expr* .(prop *id*)

  expr += *expr* .(prop *id*) <- *expr*

- Filtrage des nœuds :

  pat += node <*tag*> *pat* [prop *id* = *pat*]* end

1. Sémantique opérationnelle (à grands pas).
   - utilisation de ^{C}DOM pour les opérations sur les nœuds ;
     - résultat de `node` $\cdots$ `end` = nœud noir ;
     - imbrication correcte des portées.
   - délégation de tous les effets à ^{C}DOM.
     - données du langage = nœuds blancs ;
     - séparation contrôle/effets ;
     - spécification inhabituellement précise.
2. Système de types.
   - Typage ML classique ;
   - typage des nœuds sans l'imbrication (propriétés uniquement) ;
   - encodable dans les systèmes existants ;
   - expressivité suffisante pour typer des traitements courants.

Définitions de types de nœuds

- Définition d'un type de nœud :
  phrase += node type <*tag*> [prop *id* : *type*]* end
  - étiquette ;
  - ensemble de propriétés typées ;
  - pas (encore) de typage de la validité de l'imbrication ;
- crée un type monomorphe <*tag*> node ;
- pas de types de nœuds avec la même étiquette ;
- même propriété dans 2 types $\Rightarrow$ même type associé.

Sous-typage

- Deux types possibles pour un nœud :

  type += <*tag*> node | node

- sous typage :

  <*t*> node (type étiqueté) $<:$ node (type générique)

- opérations réduites sur node :
  - children et replace ;
  - accesseur prop? (renvoie un type option) ;
- spécialisation par filtrage.

Sélection de nœuds partageant une propriété

```
1: let rec nodes_by_class sc n =
2:   match n.(prop? class) with
3:     | Some c ->
4:       if c = sc then
5:         n :: map (nodes_by_class sc) (children n)
6:       else
7:         map (nodes_by_class sc) (children n)
8:     | None ->
9:       map (nodes_by_class sc) (children n)
```

Sélection de nœuds de même étiquette

```
1: let rec print_int_consts n =
2:  match n with
3:   | node<const> [] prop val = i end ->
4:     print_int i
5:   | _ ->
6:     iter print_int_consts (children n)
```

Récapitulatif

Sûreté d'exécution avec ce que nous avons vu :

- Typage ML classique,
- construction de nœuds respectant les définitions,
- accès/modifications de propriétés et enfants bien typés,
- pas de déplacements grâce à ^{c}DOM,
- copie utilisant la portée lexicale `node` $\cdots$ `end`,
- mais pas d'assurance de construire des documents valides.

Un nouveau modèle de document

FidoML : validité du document

Deux opérations à typer :

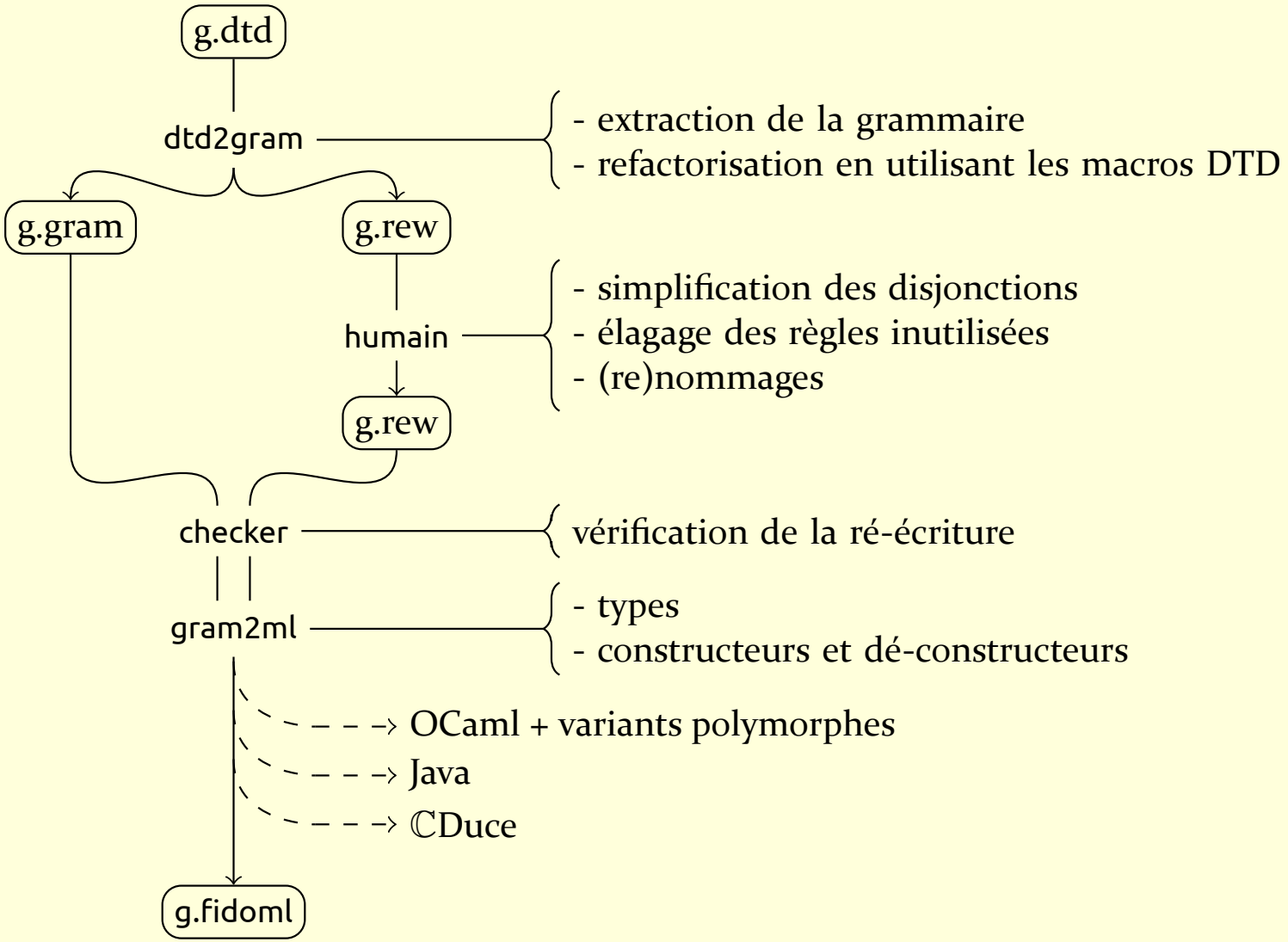
1. Création `node` $\cdots$ `end`.
2. Modification `replace`.

Il faut assurer que la séquence utilisée est conforme à la grammaire.

Deux possibilités :

- Utiliser un système de types pour XML,
- encoder la grammaire dans le système existant.

- Type encodant la séquence de nœuds (au lieu d'une liste) ;
  phrase += node type <*tag*> *type* [prop *id* : *type*]* end
- enfants = parcours en profondeur de la valeur ;
- exemples d'encodages :
  - li+ : type li_plus = node * node list
  - a | b : type a_or_b = A of <a> node | B of <b> node
- on veut un encodage automatique et sûr.


g.dtd
dtd2gram
- extraction de la grammaire
- refactorisation en utilisant les macros DTD
g.gram
g.rew
humain
- simplification des disjonctions
- élagage des règles inutilisées
- (re)nommages
g.rew
checker
vérification de la ré-écriture
gram2ml
- types
- constructeurs et dé-constructeurs
OCaml + variants polymorphes
Java
ℂDuce
g.fidoml

Récapitulatif

Sûreté d'exécution avec ce que nous avons vu :

- Typage ML classique,
- construction de nœuds respectant les définitions,
- accès/modifications de propriétés et enfants bien typés,
- pas de déplacements grâce à ^{c}DOM,
- copie utilisant la portée lexicale `node` $\cdots$ `end`,
- construction de documents valides pour une DTD,
- conservation de la validité durant les manipulations.

Conclusion

et pistes de travail

On a montré

- qu'on peut programmer le Web avec un seul langage,
- que le modèle de PHP/JavaScript n'est pas immuable,
- et qu'il est possible de typer les manipulations.

Et on a fourni

- une plateforme d'expérimentation de modèles,
- un modèle de document polyvalent et uniforme,
- un langage permettant les manipulations statiquement typées.

Vers un FidoML multi-tiers :

- manipulations distantes de document ;
- Intégration à OCaml/OBrowser.

Pistes de recherche :

- gestion mémoire dans une application Web ;
- généralisation des informations de portée :
  - envoi/migration de documents ;
    (localisation automatique client/serveur des données)
  - extension à d'autres entités.
    (sessions, données persistantes, etc.)

Merci !