Rapport de stage
Programmation Enyo en OCaml

Vincent Botbol

10 août 2012

Table des matières

Chapitre 1

Introduction

Chapitre 2

Documentation Enyo

2.1 Présentation

Enyo est un framework Javascript orienté-objet. Il privilégie un style de développement basé sur une encapsulation arborescente de composants.

Dans sa première version, Enyo fut employé pour le développement d'application des tablettes HP TouchPad fonctionnant sous webOS. Après un succès mitigé, HP décida de transformer webOS en projet *open source* devenant ainsi : Open WebOS.

Enyo passe ainsi en version 2.0. Profitant de sa conception en Javascript, l'engouement actuel de ce dernier permet aux développeurs de déployer leurs applications sur les plateformes le supportant, c'est-à-dire, tout appareil se voulant grand public (Android, iOS, WindowsPhone, etc.)

On peut attribuer à Enyo des avantages tels que sa prise en main aisée grâce au style de programmation proche d'une vision du DOM et le poids relativement faible de sa bibliothèque.

2.2 Débuter en Enyo

Il est important de distinguer deux phases dans le développement d'une application Enyo.

- La première, dite de création, consiste à définir les composants, leurs positions dans la hiérarchie de l'application ainsi que le modèle.
- La seconde est celle du déploiement. Permettant à la bibliothèque Enyo de générer, à partir du squelette décrit par la phase de création, l'objet destiné à être instancier puis afficher.

La définition des composants de l'application consiste, vulgairement, à déclarer des objets Javascript possédant des propriétés qui auront un sens au moment d'étendre l'application.

Par exemple, définissons un simple morceau de texte contenant la chaine "Bonjour monde !" :

```
1  var monApplication = {kind: "Control",
2                        name: "MonApp",
3                        content: "Bonjour monde!"}
```

Ici, nous avons simplement créé un objet Javascript muni de propriétés suffisantes pour renseigner Enyo sur le type d'objet à créer.
Explicitons ces propriétés :

- *kind* définit le type d'objet Enyo que nous souhaitons créer.
- *name* nomme l'objet que nous sommes en train de créer. C'est par la valeur de cette propriété que nous pourrons accéder à un composant ou encore l'instancier. [1]
- *content* attribut un texte au composant. Ceci n'a de sens que parce que l'on manipule un objet Enyo de type "Control" ou un de ses descendants qui possède donc cette propriété.

Désormais défini, il faut passer le composant à la méthode *kind* d'Enyo pour finaliser l'application et la rendre instanciable.

```
1  enyo.kind(monApplication);
```

A la suite de cet appel, on pourra constater qu'une nouvelle classe *MonApp* est présente dans l'environnement. Il ne reste donc plus qu'à l'instancier pour pouvoir ensuite l'afficher.

1. Il est important de noter que si placée dans un composant en racine de l'application, la propriété *name* déterminera le nom de la classe générée. Autrement, le nom d'un sous-composant servira de clé à une table d'association définie sur le composant racine de l'application servant à récupérer les divers sous-composants

```
1   var app = new MonApp();
2   app.renderInto(document.body);
```

En s'intéressant de plus près à l'objet "app", on peut constater qu'il possède un nombre important de méthodes et de propriétés que nous n'avons pas forcément définies à la création, telle que la méthode *renderInto* que nous venons d'utiliser.

Ces méthodes ont été ajoutées par induction lors de la construction de l'application par la méthode *enyo.kind*.

Si le champ *kind* spécifie un "Control", il est donc nécessaire à Enyo de lui rajouter toutes les propriétés et méthodes que possède un Control Enyo qui n'ont pas été définies par l'utilisateur.

2.3 Ajout de sous-composants

Enyo suivant une logique d'arborescence de composants. Lorsque l'on crée son application, il est nécessaire de pouvoir lui ajouter des sous-composants. On réalise cela en définissant la propriété *components* avec en valeur le tableau des composants que nous souhaitons.

Imaginons ici que l'on souhaite créer une application consistant d'un champ de saisie et d'un bouton de validation :

```
1  enyo.kind({kind:"Control",
2             name:"MonApp",
3             components: [
4                          {kind:"Input"},
5                          {kind:"Button", content:"ok"}
6                         ]
7            });
```

La fonction kind va récursivement créer tous les sous-composants de la hiérarchie ainsi que leur méthodes associées.

Comme précédemment, pour l'afficher, on instancie la nouvelle classe ainsi générée et on lui applique une méthode d'affichage :

```
1   var app = new MonCadre();
2   app.renderInto(document.body);
```

Si l'on observe le code html après ces appels, on s'aperçoit qu'enyo a injecté dans le DOM une balise html de type dans laquelle sera placée, respectivement, deux balises : un <input type="text" ..> et un <button>.

Notons que nous n'avons pas défini la propriété *name* des sous-composants. Rappelons que le nom d'un composant sert uniquement à l'accession de celui-ci par les autres composants de la hiérarchie ou encore pour son instanciation. Cependant, lors du kind, seule la racine de l'arbre sera étendue à l'environnement.

Il faut aussi noter que si la propriété name n'est pas définie, Enyo s'en charge en lui donnant un label généré ("input0" et "button0" ici). Il n'est évidemment pas recommandé d'utiliser ces derniers pour y accéder.

2.4 Événements

Reprenons l'exemple précédent et supposons désormais que l'on souhaite, lorsque l'on clique sur le bouton, afficher un message d'alerte.

```
enyo.kind({kind:"Control",
           name:"MonCadre",
           components: [
             {kind:"Input", value:""},
             {kind:"Button", content:"ok",
              handlers:{ontap:"monCallback"},
              monCallback:function(sender, event){
                alert("ok");
                return true;
              }
             }
           ]
          });
```

Au bouton, nous avons rajouter une propriété *handlers*. Cet objet sert à définir tous les événements que l'objet va traiter. Cela peut-être des événements du dom (onkeyup, onload, ...) comme des événements spécifiques à la plateforme sur laquelle l'application est déployée (l'inclinaison d'une tablette par exemple). Ici, nous utilisons l'événement "ontap", alias d'"onclick".

A chaque propriété-événement est attaché une chaine contenant le nom de la fonction à appeler.

Nous définissons donc la fonction click qui prend en premier argument l'emetteur de l'événement et l'événement envoyé contenant ses propriétés.

La valeur de retour de cette fonction spécifie si l'événement doit s'arrêter ou remonter ("bubble") aux composants parents. On retourne vrai pour l'empêcher de remonter et faux pour le laisser "bullé" sur ses ancêtres.

On notera qu'en général, les fonctions de traitement ne pas censées devoir "buller" sont définies à la racine de l'arbre qui possède un accès à tous ses sous-composants grâce à sa table de hash "$".

Il existe quelques spécificités pour la manipulation d'événements :

- Si la fonction de traitement n'est pas trouvée, l'événement est remonté dans l'arbre de composants jusqu'à trouver une fonction dont le nom correspond.
- Il n'est nécessaire de déclarer le bloc *handlers* que lorsque la fonction de traitement se situe dans le même objet que celui-ci. Par exemple, cette définition est parfaitement valable :

```
enyo.kind({kind:"Control",
           components:[{kind:"Bouton",
                        ontap:"traiteClique"}
                      ],
           traiteClique:function(sender, event){
             alert("Click recu"); return true
```

```
7                   }
8       });
```

– Il est possible de propager un événement manuellement en utilisant la fonction *send(nomEvenement, evenement)* de l'objet *enyo.Signals*. Cet événement est remis à tous les objets de type *kind :"Signals"*. Il suffit alors à l'objet-récepteur de posséder un sous-composant de ce type et d'implémenter la fonction de traitement lié au handler du sous-composant. (voir exemple)

2.5 Propriétés published

Les propriétés placées dans l'objet "published" du composant sont en général utilisées pour la logique de l'application. Certains "published" sont déjà présents dans les objets Enyo. Par exemple, la propriété "content" de l'objet "Control" est elle-même publiée.

Enyo traite ces propriétés comme étant des variables publiques et génère automatiquement : getter, setter ainsi qu'une méthode valueChanged appelée à chaque appel du set avec une nouvelle valeur.

En résumé, un composant déclaré ainsi :

```
1   {kind:"app", published:{maVariable:0}}
```

est équivalent à :

```
1   {kind:"app", maVariable:0,
2     getMaVariable:function(){..},
3     setMaVariable:function(){..},
4     maVariableChanged:function(){}}
```

Cependant il reste possible d'initialiser les valeurs de propriétés déjà publiées hors de l'objet *published* de la même façon dont nous procédions avec le champ *content* jusqu'à présent. Cela aura uniquement pour effet de masquer la première initialisation que le constructeur Enyo aura effectué.

Note : la propriété *monObjet.published.maVariable* ne sert qu'à indiquer au constructeur d'objet Enyo qu'il doit générer la variable et sa barrière d'abstraction. Des effets de bords sur cette variable après instanciation n'auront aucun effet.

2.6 Copie, partage et réutilisation de composants

Il est possible de réutiliser des composants déjà créés. Il faut cependant faire attention à ne pas lui attribuer de nom. Dans le cas échéant, le constructeur considèrera que l'objet existe déjà et tous les traitements visant une des occurences de ce nom seront appliqués au premier objet défini.

```
var unBouton = {kind:"Button", name:"bouton1",
                handlers:{ontap:"clique"},
                content:"un bouton",
                clique:function(){
                  this.setContent("Reception du click")
                }
};

enyo.kind({name:"App", components:[unBouton,unBouton]});
```

Dans cet exemple, au moment de l'instanciation un message d'avertissement concernant une collision de nom est émis mais les deux boutons s'afficheront correctement. Cependant, on constate bien que les clicks émis sur l'un ou l'autre des boutons n'auront pour effet de changer que le premier des deux boutons.

Typiquement, si l'on souhaite réutiliser un composant déjà implémenté, il est recommandé de procéder ainsi :

```
enyo.kind({kind:"Button", name:"MonBouton",
           handlers:{ontap:"clique"},
           content:"un bouton",
           clique:function(){
             this.setContent("Reception du click")
           }
});

enyo.kind({name:"App",
           components:[
             {kind:"MonBouton", name:"bouton1"},
             {kind:"MonBouton", name:"bouton2"}]
});
```

Une nouvelle classe *MonBouton* est alors étendu à l'environnement qui pourra être "Kindé" comme un objet Enyo standard dans une définition de composant.

Chapitre 3

Composants Enyo

3.1 Hiérarchie des objets

3.2 Bibliothèques additionnelles

3.2.1 Onyx

3.2.2 Canvas

3.2.3 Layout

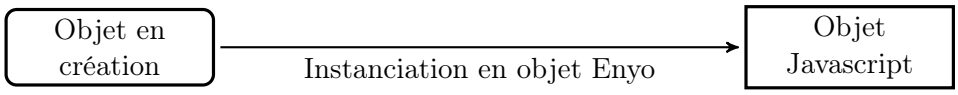
Chapitre 4

Interfaçage d'Enyo en OCaml

Pour une adaptation en OCaml d'Enyo, il est nécessaire d'établir une communication entre les deux langages. Cela présente une première difficulté étant donné le peu de points communs existants. Nous pouvons néanmoins résoudre le problème en utilisant *Js_ of_ ocaml*, un compileur de byte-code OCaml vers Javascript développé par l'équipe d'*Ocsigen*. Nous obtenons ainsi un moyen fiable d'assurer une traduction, de déclarer des objets Javascript ou encore d'attacher des méthodes à ces derniers, le tout en restant dans le monde OCaml. La transposition assurée, nous pouvons nous concentrer sur la modélisation d'Enyo.

Le point critique de l'interfaçage se situe dans l'utilisation des traits dynamiques de Javascript, par exemple, utiliser la méthode *setContent(value)* sur un *kind :"Control"* qui ne possèdera cette méthode qu'après instanciation est rédibitoire pour une transcription directe en OCaml.

Après considérations, j'ai choisi de m'arrêter sur un modèle reposant sur une différenciation d'un objet en construction de celui d'un objet instancié. On représente ainsi un objet *Enyo* en OCaml grâce aux propriétés le définissant.



L'extension de l'objet en cours de création avec propriétés et méthodes personnalisées est abandonné. Cet aspect d'Enyo est nécessaire pour la partie logique de l'application, ce dont nous pouvons nous abstraire puisque nous souhaitons un modèle de calcul reposant sur un programme OCaml classique.

A partir de là, les objets de l'interface de programmation (API) d'Enyo et leurs méthodes sont suffisants si l'on considère utiliser Enyo comme une interface graphique multi-plateforme pour OCaml. Il est également nécessaire de pouvoir traiter les événements utilisateurs pour bénificier d'une interface dynamique et réactive.

Dans un premier temps (4.1), nous présenterons la structure choisie, son fonctionnement et les raisons de cette décision.

Ensuite, la définition des propriétés et leur implémentation (4.2)

Enfin (4.3), nous nous attarderons sur les événements.

4.1 Représentation des objets

4.1.1 Modélisation

D'après le modèle choisi, on distingue donc la représentation en deux types. Le premier, à créer par l'utilisateur, que nous nommerons "kind" par la suite. Et le second, résultant d'une instanciation, possédant toutes les méthodes décrit dans l'API Enyo, que nous appellerons "obj".

Dans un soucis d'aisance, j'ai fait le choix d'annoter ces "objets" à l'aide de types fantômes. Associés aux variants polymorphes, on profite ainsi d'un sous-typage. Cela a pour avantage de permettre un partage des noms de méthodes appartenant à plusieurs objets. Le désavantage majeur de cette solution reste cependant la confusion liée à la génération des messages d'erreurs lors du développement d'une application.

Nous obtenons ainsi les deux types "kind" et "obj" qui seront utilisés avec l'ensemble de tous les type objets Enyo en précisant l'utilisation covariante du paramètre.

```
type any_id =
      [ `CONTROL | `INPUT | `BUTTON (* | ... *) ]
type +'a kind (* constraint 'a = [< any_id] *)
type +'a obj
```

Les signatures de méthodes partagées prennent alors la forme suivante :

```
sig:
(*...*)

   val setContent :
       [< `BUTTON | `CONTROL (* ... *) ] obj
       -> string
       -> unit

(*...*)
end
```

Pour construire l'objet Enyo, ce dernier doit pouvoir contenir :

- Les propriétés de bases incluent dans l'API Enyo
- Les fonctions de traitement des différents évenements
- La liste des sous-composants

Le choix s'est donc porté sur une structure de ce type :

```
type handler = Handler of string * (any_id obj -> any_id obj
   -> any_event obj -> bool)
type js_value = Int of int | String of string
```

```
                | (* ... *)
                | Array of js_value list
                | Component of any_id obj
type +'a kind = {id:string;
                 components: any_id kind list;
                 handler_list:handler list;
                 prop_list : (string * js_value) list
                }
```

J'incorpore à la structure un champ *id* pour pouvoir permettre une introspection sur les futurs objets instanciés (voir 4.3).

Quant à la représentation Javascript, elle est simplement représentée par un objet Javascript du module "Js.Unsafe" de la bibliothèque *js_of_ocaml*.

```
type +'a obj = Js.Unsafe.any
```

Cela permet un travail direct sur les différents appels à transmettre au monde Javascript.

Munis des structures, on peut désormais s'intéresser à la construction proprement dite de cette représentation OCaml d'un objet Enyo.

4.1.2 Création

Pour cette phase de création, j'ai décidé de m'inspirer du module Tk d'OCaml qui utilise les options pour les fonctions prenant un nombre important d'arguments et omissibles.

Plusieurs autres solutions auraient été possible, telle qu'une liste d'association mais dont je n'ai pas su trouver de façon élégante pour permettre à l'utilisateur de l'employer aisément.

Ainsi, nous obtenons un constructeur, ici de bouton pour l'exemple, de cette forme :

```
val button:
  ?components:any_id kind list
  -> ?content:string
  -> ?showing:bool
  -> ?src:string
  -> ?canGenerate:bool
  (* ... *)
  -> ?ontap:([`BUTTON] obj ->
             any_id obj ->
             [`GESTURE] obj -> bool)
  -> unit -> [>`BUTTON] kind
```

donnant à l'utilisation, pour un bouton contenant un simple label, ceci :

```
let mon_bouton = bouton ~content:"Valider" ()
```

Quelques bémols, cependant, à cette manière de procéder :

- Le nombre d'arguments de chaque constructeur évolue au fur et à mesure que l'objet soit profondément situé dans l'arbre des composants.

Ainsi, une feuille de cet arbre possédant une dizaine d'ancêtres possédera un nombre d'arguments important,
- Les messages d'erreurs générés deviennent rapidement indigestes,
- Une grande redondance parmi les différents constructeurs.

Finalement, le code du constructeur revient à inclure chaque argument optionnel passé dans différentes listes (typiquement, la liste de propriétés et celle de fonctions de traitement) en testant au préalable leur existence. En ajoutant la liste de composants quelques constantes, un "ID" pour l'introspection et le champ "kind" nécéssaire à Enyo, l'objet est prêt à être instancié en tant qu'application ou à être passer en tant que sous-élément dans un autre objet par sa liste de "components".

```
let button
    ?(components=[])
    ?content
    ?ontap
    (*...*)
    () =
  let prop_list= ref [("kind", String "Button")]
  and handler_list= ref [] in
  (match content with
      Some v -> prop_list := ("content",String v)
                             ::!prop_list
    | None -> ());
  (match showing with
      Some v -> prop_list := ("showing",Bool v)
                             ::!prop_list
    | None -> ());
  (*...*)
  (match ontap with
      Some v -> handler_list := Handler ("ontap",v)
                                ::!handler_list
    | None -> ());
  {id="BUTTON";
   components=components;
   prop_list=(!prop_list);
   handler_list=(!handler_list)}
```

4.1.3 Instanciation

Après avoir obtenu notre représentation OCaml, il est nécessaire de le traduire en Javascript. C'est le rôle de la fonction *Instanciate*.

Cette dernière parcourt les différentes listes (propriétés, évenements, et composants) et retourne un objet Javascript formé selon les spécifications d'Enyo.

Une application d'*instanciate* sur le "bouton kind" définit plus haut (4.1.2) retournera un équivalent en Javascript de :

```
1   {kind:"Button", content:"Valider", _onyo_id:"BUTTON"}
```

La fonction étant relativement indigeste, je détaillerai ici les manipulations éxecutées :

1. La liste de propriétés est évaluée et chaque type de valeur est traduite si besoin en Javascript. Chaque propriété est ensuite injectée dans un nouvel objet Javascript à l'aide de *Js.Unsafe.set*
2. Les différentes fonctions de traitement présentes sont "enveloppées" dans l'objet en tant que méthodes dans l'objet avec un appel à *Js.wrap_meth_callback* prenant une fonction dont le premier argument est l'objet lui-même représentant le *this*. Il est aussi nécessaire de déclarer l'objet *handlers* pour assurer le traitement des événements. (Plus de détails : 4.3).
3. Récursion sur tous les sous-composants présents et transformation de la liste de ces derniers en un tableau Javascript à passer au champ "components" de l'objet.

Voici, une forme épurée de cette fonction :

```
let instanciate (kind:([< any_id] as 'a) kind) : 'a obj =
  let rec build_component_tree : 'a. ([< any_id] as 'a) kind
      -> Js.Unsafe.any = fun kind ->
      let js_obj = Js.Unsafe.new_obj (variable "Object")
          [|||]
      in
      (* 1 *)
      ;
      (if kind.handler_list != [] then
          (* 2 *)
      );
      (if kind.components != []  then
          let array_component = Array.of_list (List.map
              build_component_tree kind.components) in
          Js.Unsafe.set js_obj "components" (array
              array_component));
      Js.Unsafe.set js_obj "_onyo_id" (string (kind.id));
      js_obj in
   kind_it (build_component_tree kind)
```

Note : il a été nécessaire d'utiliser un quantificateur universel pour unifier la récursion avec la fonction afin d'assurer le polymorphisme

Enfin, on passe cet objet à la fonction *enyo.kind* pour récupérer un objet Enyo dont les méthodes sont désormais appelables tel que *setContent* si l'objet hérite de *Control* dans la hiérarchie Enyo.

L'utilisateur est désormais en mesure d'afficher son application en effectuant un appel à la méthode *renderInto* (si l'objet en possède la méthode) sur l'objet instancié.

4.2 Gestion des propriétés et des méthodes

Pour pouvoir représenter les valeurs Javascript faiblement typées, il a été nécessaire de définir un ensemble de ces type valeurs. Un type somme est suffisant :

```
type js_value = Int of int | String of string
              | Char of char | Float of float
              | Dom_node of dom_node | Bool of bool
              | Array of js_value list | Component of any_id
                 obj
```

Note : j'ai trouvé préférable de représenter le tableau Javascript sous forme de liste, cette dernière étant plus usitée par le programmeur OCaml.

Les propriétés possédés par les objets Enyo sont implémentées en tant que "Published" (voir 2.5). Ces propriétés possèdent donc des accesseurs (get) et des modificateurs (set) ainsi que la fonction *<valuename>Changed*. Il faut alors pour chaque propriété générer ces méthodes.

J'ai fait le choix d'omettre la possibilité de définir un *<valuename>Changed* qui peut cependant, si la logique de l'application le nécessite, être remplacé par un simple appel de fonction au moment de la modification de propriété.

Au niveau de l'implémentation, j'ai ajouté ces accesseurs/modificateurs pour chaque propriété définie et appelable par tous les composants en héritant.

Afin d'assurer une bonne traduction Javascript/OCaml, il a fallu selon les types employés traduire les variables. Pour cela, l'API de "js_of_ocaml" propose les fonctions nécessaires :

```
val bool : bool -> bool t
val to_bool : bool t -> bool
(*...*)
val array : 'a array -> 'a js_array t
val to_array : 'a js_array t -> 'a array
```

Selon le type de la propriété, on s'assure alors d'effectuer correctement cette traduction :

```
let getContent this () =
  let value = Js.Unsafe.meth_call
                  this
                  "getContent"
                  [||] in (* Appel de la methode sur l'objet
                     Javascript *)
  to_string value (* converison de la valeur js obtenu en
      valeur caml *)
```

```
let setContent this chaine1 =
  let _ = Js.Unsafe.meth_call
              this
              "setContent"
              [|Js.Unsafe.inject (string chaine1)|] in (*
                  conversion caml -> javascript *)
  ()
```

Les propriétés Enyo étant pour la plupart initialisées par défaut, je n'ai pas jugé urgent de gérer les cas de valeur "null". Il faudrait, pour certains cas, fournir cette sécurité.

Une autre difficulté s'est posée concernant les méthodes polymorphiques. Une méthode de type *setProperty(name, value* est difficile à representer en OCaml, c'est pourquoi j'ai préferé laisser ce problème de côté afin de ne pas compliquer le typage.

Quelques propriétés ont ainsi été laissées de côté, comme par exemple celles prenant des objets Javascript non-Enyo en valeur.

```
1 //bounds : {left: _offsetLeft_, top: _offsetTop_, width:
      _offsetWidth_, height: _offsetHeight_}
2 this.setBounds({width: 100, height: 100}, "px");
3 this.setBounds({left:"100", top:"40", width: "10em", right: "30
      pt"});
```

Il conviendrait d'étudier chaque cas particulier et de fournir une équivalence sûre quitte à réduire les possibilités d'usage.

Enfin, certaines propriétés et arguments de méthodes peuvent changer de type selon leur usage et il n'est pas toujours agréable d'avoir à effectuer une conversion OCaml (ex : *string_of_int*) sur ces appels. Une solution pourrait être de dupliquer ces méthodes, par exemple, en les préfixant par le type. Ainsi, la propriété "value" de l'objet Enyo *Slideable* prenant un int, qui par ailleurs effectue une collision de nom avec celle de l'objet *Input* (voir 5), pourrait devenir *int_getInput* s'appliquant à l'objet Slideable et à ses descendants.

4.3 Gestion des événements

4.3.1 Fonctionnement

Dans Enyo, la spécification des événements et de leur fonctions de traitement gardent toujours la même forme.

```
1   fonctionTraitement: function(emetteur, evenement){
2                           //traitement
3                           return bool; }
```

La fonction attachée prend en paramètres l'émetteur de cet événement, c'est-à-dire, le composant qui a genéré ou propagé celui-ci, et l'objet évé-

nement lui-même. Celui-ci retourne un booléen spécifiant si la propagation doit être effectué. (voir : 2.4 pour plus de détails)

Pour adapter l'évenement en OCaml, j'ai choisi ce type :

```
type any_event = [`GESTURE (* | Ensemble des evenements *) ]
type handler = Handler of string * (any_id obj -> any_id obj
    -> any_event obj -> bool)
```

La gestion de l'événement est donc représentée par un couple contenant le nom de l'événement et la fonction de traitement associée avec en premier argument le *this* de la méthode (voir : 2).

En informant la signature dans chaque constructeur, il devient aisé de constituer un typage correct. Ainsi, nous pouvons résoudre le *this* selon l'objet en cours de construction et l'événement attendu. L'émetteur ne peut cependant pas être résolu, nous y reviendrons plus loin.

Pour un bouton, nous attendrons un argument de ce type :

```
val button:
  (* ... *)
  -> ?ontap:([`BUTTON] obj -> any_id obj -> [`GESTURE] obj ->
      bool)
  -> unit -> [>`BUTTON] kind
```

A l'utilisation, nous pourrons alors utiliser les méthodes de button sans causer de conflit de type puisqu'employé dans un contexte correct :

```
let traitement_bouton this emetteur evenement =
   setContent this "Nouveau_contenu";
   setDisabled this true;
   true

let mon_bouton = button ~content:"contenu" ~ontap:
    traitement_bouton ()
```

J'ai défini les événements de la même manière que les objets, à ceci près qu'il n'est pas nécessaire d'avoir plusieurs méthodes par événements puisque les événements ne sont pas liés entre eux. Il est cependant nécessaire d'affecter à chaque type d'événements des fonctions d'accesseurs pour récupérer les informations contenus.

Par exemple, pour l'événement souris, rebaptisé "gesture" pour se conformer à Enyo, les accesseurs, stipulés dans la définition IDL W3C, sont définis :

```
val gesture_screenX : [`GESTURE] obj-> int
val gesture_screenY : [`GESTURE] obj-> int
val gesture_identifier : [`GESTURE] obj-> int
val gesture_detail : [`GESTURE] obj-> int
(* ... *)
```

Á l'instanciation d'un objet pourvu d'un événement, il est nécessaire de passer l'événement traité dans un objet contenu dans le champ handler de celui-ci. Ainsi, le *mon_bouton* définit plus haut, aura cette forme en Javascript :

```
1   {kind:"Button", handlers:{ontap:"tap"},
2     tap:function(){ return callback_caml(this) /* gere par
          js_of_ocaml */ }}
```

Cette stratégie permet à chaque composant de posséder sa fonction de traitement mais présente un gros soucis lors du "bubbling" (remontée d'événement). En effet, il faut pouvoir connaître l'origine de l'événement à tout moment .

Prenons une situation d'exemple en Enyo avec notre représentation :

```
1  enyo.kind({
2      kind:"Control",
3      name:"App",
4      components:[
5        {
6          kind:"Control",
7          components:[
8            {kind:"Button", content:"Vert",
9              handlers:{ontap:"tap"},
10             tap:function(emetteur, event){
11               //emetteur = this
12               return false; //propagation au noeud-parent
13             }
14           },
15           {kind:"Button", content:"Bleu",
16             handlers:{ontap:"tap"},
17             tap:function(emetteur, event){
18               //emetteur = this
19               return false; //propagation au noeud-parent
20             }
21           }
22         ]
23         }
24       ],
25     handlers:{ontap:"tap"},
26     tap:function(emetteur, event){
27       //emetteur = sous-control
28       // change le fond si emetteur est b1 ou b2
29       alert(emetteur);
30       return true;
31     }
32 })
```

Ici, nous voulons, si l'on clique sur l'un des deux boutons, changer la couleur du fond du "control" racine selon le bouton cliqué.

Il faut donc pouvoir tester au niveau racine, quelle a été l'origine des événements. Le problème ici provient de la propagation au composant parent par les boutons. En effet, l'argument *emetteur* reçu par la racine ne contient pas l'un des deux boutons mais le "Control" intermédiaire par lequel l'événement a dû passé.

En Enyo, un simple : *emetteur.originator* renverrait le bouton ayant généré l'événement, et nous permettrait de déterminer la source mais l'im-

plémentation de cette propriété aurait peu de sens dans la représentation actuelle.

Une solution à ce problème peut être de conserver, dans son application, une valeur référencé contenant l'objet ou l'information nécessaire pour effectuer le traitement souhaité. Je propose une autre stratégie plus loin : 4.3.3.

4.3.2 Introspection

L'implémentation actuelle de cette stratégie de traitement pose un problème pour typer l'emetteur. Pour pallier à celà, j'ai fourni lors de la construction des objets un champ supplémentaire [1] à chaque "kind" qui va s'étendre à l'objet Javascript : *_onyo_id*.

Munis de cette propriété, nous pouvons désormais accéder à ce champ pour le typer correctement. C'est le rôle de la fonction *as_a* :

```
val as_a : ([< any_id] as 'a) -> [< any_id] obj -> 'a obj
```

Si l'objet n'est pas de l'"id" spécifié, une exception est levé.

Avec ceci, nous pouvons agir sur l'emetteur :

```
let mon_bouton = ~content:"Bouton" ~tap:(fun _ _ _ -> false)
    ()

let traitement this emetteur evenement =
   try
       let bouton_emetteur = as_a `BUTTON emetteur in
       (* bouton_emetteur : [`BUTTON] obj *)
       setDisabled bouton_emetteur true;
       true
   with Bad_kind -> true

let mon_control = ~components:[mon_bouton] ~ontap:traitement
    ()
```

Cette solution reste cependant inélégante et assez contraignante pour le développeur.

4.3.3 Autre stratégie de propagation

Dans un développement Enyo, il est courant de ne définir les fonctions de traitement qu'à la racine de l'arbre de composants. Ceci permet d'avoir un accès à tous les sous-composants grâce à la table de hash que possède le composant principal. Nous obtenons ainsi le programme reprenant l'exemple posant problème :

```
1  enyo.kind({
2    kind:"Control",
```

1. Le champ kind étant effacé à l'instanciation Enyo, il n'est pas possible d'en faire usage

```
3       name:"App",
4       components:[
5         {
6           kind:"Control",
7           components:[
8             {kind:"Button", name:"b1", content:"Vert",
9               ontap:"tap"
10            },
11            {kind:"Button", name:"b2", content:"Bleu",
12              ontap:"tap"
13            }
14          ]
15        }
16      ],
17      tap:function(emetteur, event){
18        if (this.$.b1 == emetteur){
19          (* couleur verte *)
20        } else if (this.$.b2 == emetteur){
21          (* couleur bleu*)
22        }
23        return true;
24      }
25  })
```

Un comportement semblable pourrait être implémenté résolvant par la même occasion, le typage de l'émetteur.

Plutôt que de définir les fonctions de traitement au niveau des objets, on pourrait les définir sous des labels uniques, à la racine. Nous obtiendrions, une signature dans le constructeur pour le traitement de ce type ressemblant à :

```
button ?content:string
       (* ... *)
       ?ontap:([`APPLICATION] obj ->
               [`BUTTON] obj ->
               [`GESTURE] obj -> bool)
```

Il conviendrait à priori de définir un nouvel objet "Application" sur lequel il serait possible de fournir une lecture de la table de hash et ainsi d'agir sur les différents objets.

L'instanciation à même titre retournerait un : *[`APPLICATION] obj.*

J'essayerai de convenir de la validité d'une telle approche et de son implémentation dans une version future.

Chapitre 5

Langage de Description d'Interface (IDL)

Enyo fournit une API suffisamment grande pour qu'il devienne rébarbatif d'implémenter ceci à la main. L'implémentation d'un IDL s'est donc révelé être une bonne option pour pouvoir représenter facilement l'arbre de composant.

OCaml étant un outil tout indiqué pour ce type de travail, je n'ai pas eu à faire de parallèle avec d'autre langage.

En premier lieu, il convient de présenter les structures choisis pour abriter les objets. Puis, j'expliquerai brièvement la génération du code et les problèmes rencontrés.

5.1 Représentation des objets

Un objet Enyo présente 3 grandes parties distinctes :
- Les méthodes,
- les propriétés,
- les événements qu'il est capable de "capturer".

Pour les méthodes et les propriétés, il est nécessaire de fournir le type à l'IDL pour assurer à OCaml la sûreté du typage.

Les événements, quant à eux, possèdent un lien vers le type d'événement et les champs qu'il faut préciser.

```
type values_rep =  String | Int | Float | Bool | Unit
                 | Component | Array of values_rep | Dom_node

type method_rep = Method of string * values_rep list

type attributes_rep = Attribute of string * values_rep * bool (*
    bool => Generate propChanged function? *)

type event_rep = Event of string * attributes_rep list
```

```
type handler_rep = Handler of string * event_rep

type object_rep = Type of string * method_rep list *
    attributes_rep list * handler_rep list
                  | Type_gen of object_rep * string * method_rep
                      list * attributes_rep list * handler_rep
                     list
                  (* Le deuxieme element de la somme n'a plus de
                       reel utilite, mais n'ayant pas eu le temps
                  de retravailler le code, il est a ignorer *)
```

J'ai donc choisi de représenter l'objet Enyo par :

- Son nom Enyo,
- une liste de représentation de méthodes,
- une liste de représentation de propriétés [1],
- une liste des noms d'événements que ce type d'objet est capable de traiter.

On obtient ainsi pour l'objet "Control" d'Enyo, la représentation suivante :

```
Type ("Control",
      [
        Method ("render", [Unit; Unit]);
        Method ("rendered", [Unit; Unit]);
        Method ("renderInto", [Dom_node; Unit]);
        Method ("setBounds", [Int; Int; Int; Int; Unit]);
        Method ("show", [Bool; Unit]);
        Method ("write", [Unit; Unit]);
        (*...*)
      ],
      [
        Attribute ("style", String, true);
        Attribute ("content", String, true);
        Attribute ("allowHtml", Bool, false);
        Attribute ("src", String, true);
        (*...*)
      ],
      [Handler ("ontap", Gesture_event._gesture)]
)
```

En étudiant l'API, j'ai pu transcrire les méthodes et propriétés en m'efforçant de typer au plus juste. En revanche, quelques libertés ont été prises :

- Je n'ai pas jugé utile l'ajout de méthodes trop polymorphes telle que : *setProperty(name, value)*.
- Par manque de documentation de l'API sur certaines, il a été quelques fois nécessaire de compléter moi-même les signatures de méthodes en effectuant des tests en restant toutefois rigoureux.

1. L'utilisation du booléen pour savoir si une fonction de type <value>Changed doit être implémenté est temporairement abandonné, pour les raisons spécifiées dans la partie de gestion des propriétés 4.2

– J'ai omis d'inclure les méthodes protégées, pour la simple raison que nous ne traitons pas l'extension objet, ici.

Je complète ainsi chaque objet et l'inclus dans un arbre d'"object_rep" recréant ainsi la hiérarchie Enyo nécessaire permettant à chaque composant d'hériter des méthodes et attributs. La hiérarchie désormais complète, la génération de code peut débuter.

5.2 Générateur de code

L'objectif de cette génération de code est d'obtenir en sortie un module complet basé sur la représentation IDL avec le modèle choisi 4.

La difficulté de cette étape a été le parcours de l'arbre pour résoudre les dépendances :

– Un objet doit transmettre à ses fils toutes les propriétés et "handlers" d'événements qu'il possède afin que leurs constructeurs soient en mesure de les proposer à l'utilisateur
– En revanche, les méthodes auront besoin de la liste de tous les identifiants de sa descendance pour pouvoir spécifier les méthodes et permettre aux enfants de les appeller

Une simple récursion sur l'arbre permet l'obtention d'un résultat satisfaisant.

Plus tard, j'ai constaté qu'un cas spécial ne permettait pas un parcours aussi banal : lorsque deux objets définissent des méthodes ou proprieté de même noms sur des branches différentes de l'arbre, il y a collision de noms et il est impossible de factoriser simplement les dépendances [2].

Afin de résoudre ceci, je réalise des parcours préalables en remplissant des tables de hachage contenant en clé le nom de méthodes et de proprietés avec en valeur la liste des dépendances. Puis j'applique la récursion, en consultant la table de hachage pour résoudre les incohérences.

Note : On aurait également pu "tagger" les différentes incohérences au moment de la représentation de l'objet en spécifiant que cette méthode ou propriété est partagée avec un autre objet.

Mon arbre de résolutions de dépendances construit, il ne reste plus qu'à appliquer la mise-en-forme du module pour pouvoir imprimer le code OCaml.

2. J'omets le fait que deux méthodes de même nom peuvent être de types différents, ce qu'il faudrait traiter si le cas était présent

Chapitre 6

Applications

6.1 Déploiements

Chapitre 7

Conclusion

Chapitre 8

Annexes

Chapitre 9

Bibliographie