Applications du *toplevel* embarqué d'Objective Caml

Clément Capel[1,2] & Emmanuel Chailloux[1] & Jean-Marc Eber[2]

1 *Equipe Preuves, Programmes et Systèmes (UMR 7126),*
Université Paris 6,
175, rue du Chevaleret, 75013 Paris, France
`Clement.Capel@pps.jussieu.fr`
`Emmanuel.Chailloux@pps.jussieu.fr`

2 *LexiFi SAS, INRIA, Bâtiment 1*
Domaine de Voluceau-Rocquencourt, B.P. 105
78153 Le Chesnay Cedex, France
`jeanmarc.eber@lexifi.com`

Résumé

On présente dans cet article les modifications à apporter à la distribution Objective Caml pour construire son *toplevel* sous forme d'une bibliothèque. Cela autorise de l'embarquer dans une application pour pouvoir exécuter à la volée des fragments de programmes Objective Caml. On illustre l'intérêt, tant applicatif que pédagogique, de cette démarche en construisant deux *plugins* d'applications grand public. Le premier permet de télécharger des programmes Objective Caml et de les exécuter dans la fenêtre graphique d'un navigateur Web. Le deuxième s'intègre dans Excel comme langage de script du langage de macros Visual Basic. L'exécution d'une macro VB produit un programme Objective Caml dont l'évaluation retourne le résultat directement à la feuille de calcul. Ces deux exemples permettent d'apprécier la puissance d'expression et la compilation efficace d'Objective Caml qui en fait un candidat pertinent comme langage embarqué.

Introduction

Il n'est pas rare qu'un logiciel ait besoin au cours de son exécution d'évaluer des fragments de programmes issus d'un autre langage que celui de son implantation. On parle alors de langages de «glue», de «macros», de «scripts» ou de langages embarqués.

Cela est particulièrement utile pour la paramétrisation de l'application mais aussi pour l'interaction avec l'utilisateur. On peut citer un certain nombre de langages de script comme `sh` pour Unix, `Lisp` pour `Emacs` et `Gimp`, `Visual Basic` dans Excel, `javascript` pour les page Web du coté client et `Php` ou `perl` du coté serveur, `Lua`[1] ou `VSL`[2] pour les jeux vidéo, etc. Dans d'autres cas, c'est le programme lui-même qui engendrera le texte source à évaluer, par exemple pour la construction d'une requête `SQL`. Ces langages sont plus faciles d'utilisation que l'emploi de la bibliothèque de l'application tout en ayant accès à l'environnement de celle-ci. Ils sont le plus souvent interprétés. L'interpréteur embarqué dans une application est lié aux fonctions de celle-ci tout en sachant communiquer des valeurs de son format à celui de l'application. La genèse de ces langages répond souvent à un besoin immédiat. En cas de succès, ces langages ont tendance à devenir plus importants en incorporant de nouvelles fonctionnalités. Il en est de même des programmes écrits avec ceux-ci qui, souvent, grossissent fortement. On rencontre alors des difficultés liées à des sémantiques alourdies par la compatibilité ascendante et à de piètres

[1] `http://www.lua.org`
[2] `http://www.virtools.com`

efficacités dues en partie à l'interprétation. Par exemple le programme `latex2html`[3] écrit en `perl` dépassait les dix mille lignes et avait de réelles lenteurs et difficultés pour traduire des documents importants. Certains de ces langages sont compilés vers du code-octet. Celui-ci peut être ensuite interprété par une machine virtuelle embarquée (`Lua`, `VSL`). L'intérêt est alors un gain d'efficacité à l'exécution.

Le mécanisme d'embarquement d'un langage est proche de celui de la boucle d'interaction (*toplevel* ou boucle *Print-Eval-Read*). La partie lecture peut être interactive ou non; la partie évaluation est quasiment identique; la partie affichage devient la transmission d'une donnée dans un format accessible. Comme l'évaluation d'une phrase peut provoquer une erreur, celle-ci doit être détectable et manipulable dans l'application. La nature du typage du langage (non typé, typé dynamiquement, typé statiquement) influe sur la détection de certaines erreurs. Dans le cas de code compilé le type de chargement (statique ou dynamique) du code influence aussi l'intégration dans une application. Enfin un dernier critère est celui de légèreté de l'embarquement.

Il n'est pas surprenant de retrouver l'ancêtre des langages fonctionnels comme langage de macros dans des applications bien connues comme Emacs et Gimp. La tradition de la boucle Print-Eval-Read vient des premiers interprètes `Lisp`. De plus ce langage d'une clarté agréable a un pouvoir d'expression impressionnant. Même un langage comme `javascript` s'en inspire avec la construction `function`. La vague `Java` a proposé d'embarquer une machine virtuelle munie d'un mécanisme de chargement dynamique des classes. Mais il manque la possibilité d'exécuter des programmes `Java` source. Rien n'empêche d'embarquer un interprète `Java` comme «dynamic Java»[4] et même un compilateur `Java` comme «janino»[5] autorisant ainsi l'exécution à la volée de textes `Java`. L'expérience du langage `Scol`[6][1] montre les possibilités de chargement dynamique dans un langage statiquement typé. La plate-forme .NET [7], qui se veut un creuset où s'intègrent différents langages, possède dans son *API Reflection* les méthodes pour engendrer et charger dynamiquement du code-octet.

L'importance que peuvent prendre ces langages dans une application nécessite des précautions lors de leur conception. On rejoint alors les problématiques d'expressivité, de sémantique claire et de compilation efficace des langages de programmation. Pourquoi alors ne pas choisir un langage existant ayant ces bonnes caractéristiques? Nous nous intéressons à l'utilisation du langage Objective Caml dans ce cadre. Objective Caml [5] est un langage issu de la déjà longue lignée de la famille ML. C'est un langage fonctionnel/impératif, statiquement typé, polymorphe paramétrique, muni d'un mécanisme d'exception, possédant une extension objet et un système de modules paramétrés. Son pouvoir d'expression est important, sa sémantique de par sa tradition est formalisée et de plus son implantation, y compris le compilateur de code-octet, produit des programmes efficaces. Sa boucle d'interaction est utilisée comme outil de développement. Bien qu'une bibliothèque Objective Caml puisse être utilisée dans un programme `C` il n'est pas possible d'y embarquer le *toplevel*.

On décrit dans cet article les modifications à apporter à la distribution Objective Caml pour construire son *toplevel* sous la forme d'une bibliothèque utilisable dans un programme `C`. Cela nous autorise à exécuter des expressions ou des déclarations Objective Caml à la volée. Les communications de valeurs entre application et *toplevel* peuvent être effectuées dans le format interne d'Objective Caml ou bien dans le format binaire COM de Microsoft. Deux types d'applications sont ensuite présentées. La première application est un *plugin* pour les navigateurs Web permettant d'évaluer des programmes Objective Caml et de les visualiser dans la fenêtre du navigateur. La deuxième application est un module pour le tableur Excel autorisant l'évaluation de code Objective Caml engendré par des macros Visual Basic et dont le résultat est retourné sur la feuille de calcul.

[3] `http://www.latex2html.org/`
[4] `http://koala.ilog.fr/djava`
[5] `http://www.janino.net`
[6] `http://www.scol-technologies.org`
[7] `http://www.microsoft.com/net`

1. Embarquement du *Toplevel* Objective Caml

Objective Caml possède une interface simple avec `C` permettant de construire des applications à partir de ces deux langages. Cette possibilité inclut le code Objective Caml compilé vers du code-octet grâce à l'implantation de sa machine virtuelle en `C`. On peut ainsi évaluer des fermetures (valeurs fonctionnelles Objective Caml) à un ou plusieurs arguments et récupérer la valeur de leur application en `C`. On manipule les valeurs Objective Caml en `C` par un ensemble de fonctions et de macros.

Notre but est de pouvoir appeler le *toplevel* Objective Caml en `C`. Cela nécessite une extension légère et précise du compilateur[8]. On rappelle comment un programme `C` peut appeler des fonctions Objective Caml. On détaille ensuite les modifications apportées à la distribution Objective Caml pour qu'un programme `C` puisse manipuler le *toplevel* Objective Caml. On illustre ce propos par de petits exemples.

1.1. Interface avec C et *toplevel*

On utilise dans la présentation de l'interface avec `C` des variations du programme jouet suivant qui définit une fonction successeur :

succ.ml

```
let succ x = x + 1;;
let a = 2 in succ a;;
```

structure d'un exécutable Lorsque l'on réalise l'édition de liens d'un tel programme[9], le compilateur de code-octet construit un fichier exécutable possédant une structure particulière, comme décrit à la figure 1. L'ensemble des informations utiles à l'exécution du programme sont recensées. Parmi elles, on peut citer la table des symboles (qui stocke les noms des modules chargés, les exceptions ...) ainsi que la table des primitives (qui stocke les fonctions `C` utilisées par le programme).

RNTM	:	chemin du *runtime* (si il n'est pas concaténé)
CODE	:	instructions du programme
DLPT	:	chemins d'accès aux bibliothèques dynamiques
DLLS	:	bibliothèques dynamiques utilisées dans le programme
PRIM	:	liste des primitives utilisées dans le programme
DATA	:	données du programme
SYMB	:	table des symboles
DBUG	:	informations de *debug*
TOC	:	table des matières des différentes sections

Figure 1: structure en section et table des matières de l'exécutable Objective Caml

interface avec C Par ailleurs, il est possible d'appeler une fermeture Objective Caml depuis `C` à travers l'interface externe (figure 2).

Dans ce cas l'édition de liens n'est pas réalisée de la façon précédente. En effet, pour être appelée en `C`, une fermeture Objective Caml doit avoir été stockée dans un fichier et compilée par le compilateur Objective Caml avec l'option `"-output-obj"`. Comme la cible devra être liée, ceci impose la génération

[8] effectuée pour la distribution 3.06.

[9] `ocamlc -o succ succ.ml`

d'un fichier C, rassemblant les mêmes informations qu'un exécutable Objective Caml dans un format valide pour un compilateur C.

f.ml
```
let f x = x + 1;;
Callback.register "f_caml" f;;
```

main.c
```
value param = Val_long(2);
value v = callback(*caml_named_value("f_caml"), param);
int succ_2 = Long_val(v);
```

ocamlc -output-obj -o camlprog.o f.ml	gcc -o main main.c camlprog.o -lcamlrun

Figure 2: appel d'une fermeture Objective Caml en C / utilisation de l'option `"-output-obj"`

Afin de stocker ces informations dans un fichier C, chaque section est convertie en un tableau C (figure 3). Toutefois, seules les sections correspondant au code, aux données et aux primitives sont conservées dans le programme C.

a.o.c
```
static int caml_code[] = {0x00000054, 0x00000390, ...};
static char caml_data[] = {132, 149, 166, 190, ...};
primitive builtin_cprim[] = { array_get, array_set_addr, ... };
char * names_of_builtin_cprim[] = "array_get","array_set_addr", ... };
void caml_startup(char** argv){...}
```

Figure 3: fichier C engendré par `"-output-obj"` qui sera lié au code C existant

Lors de l'exécution du programme C, l'appel à la fonction `caml_startup` de la figure 3, située dans le fichier engendré par `"-output-obj"`, permet d'appeler la fonction d'initialisation de la machine virtuelle avec les tableaux de code, de données et de primitives. Les fermetures sont alors ajoutées à l'environnement d'exécution, et les fonctions exportables vers C recensées.

manipulation du *toplevel* par un programme Objective Caml Comme le *toplevel* Objective Caml est entièrement écrit en Objective Caml, il est possible de l'appeler au sein même d'un programme Objective Caml en le liant avec la bibliothèque `toplevellib.cma`. Ceci permet alors d'évaluer du code créé à l'exécution par le programme Objective Caml lui-même. Il est ainsi possible de définir une fonction `eval` (figure 4) pour évaluer des textes Objective Caml (expressions ou déclarations).

maniptop.ml
```
Toploop.initialize_toplevel_env();;
let eval txt = let lb = (Lexing.from_string txt) in
  let phr = !Toploop.parse_toplevel_phrase lb in
  Toploop.execute_phrase true Format.std_formatter phr;;

eval "let add1 x = x +1;;";;
eval "add1 2;;";;
```

chargement dans le *toplevel*
```
# #use "maniptop.ml";;
- : unit = ()
val eval : string -> bool = <fun>
val add1 : int -> int = <fun>
- : bool = true
- : int = 3
- : bool = true
```

édition de liens
```
ocamlc -custom toplevellib.cma maniptop.ml -o manip
```

exécution
```
> ./manip
val add1 : int -> int = <fun>
- : int = 3
```

Figure 4: le *toplevel* manipulé depuis un programme Objective Caml

impossibilité de manipuler le *toplevel* en C Il n'est malheureusement pas possible de manipuler ou d'exécuter le programme `maniptop.ml` en C. Lors de son initialisation (fonction `Symtable.init_toplevel`) le *toplevel* Objective Caml doit initialiser l'environnement du compilateur (écrit en Objective Caml) avec la liste des primitives et la table des symboles. De fait, comme la table des symboles n'est pas disponible à l'exécution (figure 3), toute tentative d'appeler le *toplevel* Objective Caml en C conduit à une exception. En outre, si les primitives sont disponibles dans l'environnement de la machine virtuelle, elles ne le sont pas dans celui du compilateur, rendant indisponible la compilation de fermeture liée à du code C et donc une grande partie de la bibliothèque standard.

1.2. Création d'une bibliothèque *toplevel*

Le but est de rendre disponible le *toplevel* sous forme d'une bibliothèque en C. Des fonctions d'accès simple sont donc proposées pour permettre de communiquer avec la boucle d'interaction, qui s'exécute bien sûr dans le même processus que le programme C appelant. L'utilisation est identique au *toplevel* classique et autorise toutes les directives, déclarations et options disponibles habituellement. Deux fonctions principales sont fournies :

- appel préalable de `toplevel_init` (1 seule fois);
- appels de `toplevel_exec` (à chaque exécution d'une phrase);

et l'édition de liens s'effectuera par un ordre de la forme suivante :

```
gcc -o nom_executable fichiers_c -ltoplevel
```

ajout de la table des symboles Pour cela, il s'agit donc, d'une part de stocker la table des symboles dans le fichier engendré par `"-output-obj"` et d'autre part de transmettre les informations de primitives et de table des symboles de la machine virtuelle (en C) au compilateur (en Objective Caml). Pour obtenir une cohérence du fichier engendré par `"-output-obj"`, la table des symboles doit être écrite dans un format C. On utilise alors le module `Marshal` pour ranger la table des symboles dans un tableau C d'entiers.

ajout des primitives Les primitives sont en fait des fonctions C qui ont la particularité d'être appelables par du code Objective Caml. Elles font partie du runtime `ocamlrun`[10] (figure 5 (a)). Afin de pouvoir être exécutées par l'interprète (figure 5 (b)), elles bénéficient d'un traitement spécial pour être connues à la fois de la machine virtuelle et du code interprété (figure 5 (c))

Dans notre cas, il faut que la table des symboles soit accessible du code qui va être exécuté (fonction d'initialisation du *toplevel*). On a donc créé une primitive pour réaliser ce transfert. En outre, comme la liste des primitives est aussi récupérée par `init_toplevel` (figure 5 (d)), elle doit subir le même type de transfert.

Une primitive est donc ajoutée, `get_symtable`, renvoyant une chaîne de caractères Objective Caml. Cette chaîne est en fait la linéarisation de la table des symboles contenue dans le fichier engendré par `"-output-obj"`, table transmise lors de l'initialisation du *runtime*. Pour que le programme Objective Caml accède à cette primitive, une interface est déclarée, se chargeant de l'appel de primitive (figure 5 (e)).

Il suffit alors au programme Objective Caml d'appeler la fonction de l'interface pour récupérer cette chaîne. Afin de récupérer la liste des primitives (figure 5 (f)), le même traitement est réalisé. Il est alors possible d'initialiser le *toplevel* avec toutes les informations nécessaires.

chargement dynamique Cependant, cela ne suffit pas pour réaliser tout chargement dynamique dans le *toplevel* embarqué. En effet, le *runtime* ne considère pas le cas d'un *toplevel* Objective Caml

[10] elles peuvent aussi être dans une bibliothèque dynamique liée à une bibliothèque Objective Caml (`threads.cma`, `graphics.cma`).

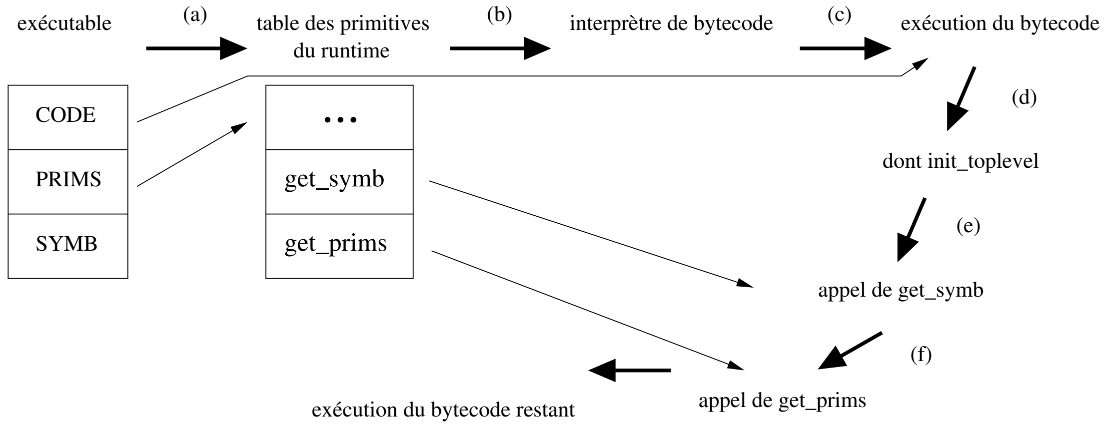

Figure 5: traitement de l'appel de primitive

appelé depuis C. De ce fait, la table des primitives du runtime est initialisée de façon complètement statique, il n'est donc pas possible de rajouter des primitives C (cas de la bibliothèque graphique). Pour cela, il faut donc réécrire l'initialisation de cette table en utilisant les fonctions d'accès, tout en maintenant la cohérence de la table lorsqu'elle est enrichie.

interface Afin, de pouvoir manipuler ce *toplevel* embarqué de façon simple, une interface est fournie.

Du côté Objective Caml, l'initialisation prend en charge l'analyse des arguments, de la même façon que l'exécutable `ocaml`. Il est donc par exemple possible de préciser des chemins de chargement et/ou des fichiers (sources ou compilés) à charger pour définir l'environnement initial d'exécution.

Du côté C, outre la gestion du passage d'argument entre C et Objective Caml, l'environnement de la console est pris en charge. Ceci permet en particulier d'utiliser les *threads*, qui vont faire passer la console en mode non-bloquant, la bibliothèque se chargeant de repasser en mode bloquant afin de permettre une éventuelle saisie au clavier en C.

Afin d'assurer le maximum d'expressivité, 3 fonctions d'exécutions sont proposées :

- `char* schema_toplevel_exec(char* code)` : retourne la sortie classique du *toplevel* (type + valeur);
- `value toplevel_exec(char* code)` : retourne la valeur de l'expression évaluée (ou une exception Objective Caml);
- `VARIANT com_toplevel_exec(char* code)` : retourne un variant COM.

1.3. Exemple

Il est maintenant possible de manipuler le *toplevel* Objective Caml en C. Voici un exemple simple de calcul de la factorielle par ce *toplevel* embarqué :

programme C

```
toplevel_init(argv);

toplevel_exec("let rec fac x = match x with 0 -> 1 | 1 -> 1 | x -> x*fac (x-1);;");
val = toplevel_exec("fac 5;;");
res = 1+Long_val(val);
printf ("1+5!=%dn", res);
```

2. *Plugin* Web pour des applications CAML

Une des motivations de l'embarquement du *toplevel* a été de construire un *plugin* d'exécution de programmes Objective Caml pour les navigateurs *web* qui soit multi-systèmes. Les navigateurs *web* sont des programmes largement utilisés dont l'interface, bien que pauvre, est bien acceptée. Il n'est pas déroutant pour un utilisateur *lambda* d'exécuter un programme comme une *applet* `JAVA` dans une fenêtre de son navigateur. Les fonctionnalités d'un navigateur peuvent être étendues pour traiter de nouveaux types de données ou de programmes grâce à un système d'extensions intégré ou *plugin*. Un *plugin* pour un langage devient alors un vecteur de diffusion simplifiée pour les applications écrites dans ce langage et contribue au succès de ce dernier.

En 1996, Francois Rouaix a développé un navigateur WWW, nommé MMM[7] et prononcé Meuh, pour la lecture de pages HTML et l'exécution d'*applets* CAML. Notre démarche est différente dans la mesure où l'on cherche à étendre des navigateurs existants grand public comme Netscape. L'idée est alors d'embarquer le *toplevel* Objective Caml dans un *plugin* Netscape [6]. Le programme Objective Caml téléchargé sera transmis au *toplevel* qui exécutera l'application graphique dans la fenêtre du navigateur.

Cette idée avait été expérimenté lors d'un TER[11][9] en lançant pour chaque programme graphique Objective Caml (utilisant la bibliothèque `graphics`) téléchargé un processus *toplevel* qui communiquait avec la fenêtre du navigateur. Cela fonctionnait sous X-window dans la mesure où une application peut utiliser une fenêtre graphique d'une autre application si elle connaît l'identificateur de la fenêtre concernée. Mais pour d'autres systèmes de fenêtrage comme Windows ou MacOSX cela n'est pas possible. En revanche un *toplevel* embarqué dans un *plugin* pourra accéder et dessiner dans une fenêtre du navigateur.

Pour apprécier cette argumentation, on décrit tout d'abord le mécanisme de base de l'exécution d'un *plugin* pour ensuite montrer comment le *toplevel* peut interagir via la bibliothèque `graphics` avec une fenêtre d'un navigateur. On illustre cette réalisation par des applications existantes n'ayant pas eu besoin d'être modifiées. On discute ensuite des choix d'implantation et des limitations de cette première intégration.

2.1. Développement d'un *plugin* pour Netscape

Le navigateur Netscape fournit une bibliothèque de développement [6] pour la réalisation de *plugins*. De nombreux autres navigateurs, comme Mozilla, Konqueror ou Safari, utilisent cette approche à l'exception récente et notable d'Internet Explorer.

Un *plugin* peut dessiner dans une fenêtre à part ou dans une partie de la fenêtre du navigateur. On s'intéresse à ce dernier cas.

On indique la présence d'un *plugin* dans une page HTML par une balise `EMBED` (ou `OBJECT`). Le source suivant indique d'utiliser un *plugin* traitant du type `x-caml-plugin` dans la page du navigateur dans une fenêtre de taille 400x300 en lui envoyant le flux correspondant au fichier `colwheel.ml`:

colwheel.html

```
<EMBED type=application/x-caml-plugin
  src="http://www.pps.jussieu.fr/~emmanuel/colwheel.ml" width=400 height=300>
```

Modèle d'exécution d'un *plugin* Au démarrage d'un navigateur, celui-ci liste les *plugins* disponibles et les enregistre en fonction des types MIME qu'ils traitent. Quand le navigateur rencontre des données d'un type MIME enregistré par un *plugin*, il charge dynamiquement le *plugin* en mémoire (s'il n'est pas déjà chargé) et crée une nouvelle instance du *plugin*. Il peut avoir plusieurs instances d'un *plugin* à un moment donné (plusieurs balises EMBED dans une même page ou plusieurs pages

[11]Travaux Encadrés de Recherche

avec une telle balise). Quand l'utilisateur quitte la page ou détruit la fenêtre l'instance du *plugin* est détruite. Quand toutes les instances d'un *plugin* sont détruites, celui-ci est libéré de la mémoire.

Lors de la création d'une instance d'un *plugin* le navigateur lui indique le descripteur de la fenêtre graphique et si besoin est crée un flot de communication pour transmettre les données à traiter. Le *plugin* doit pouvoir répondre aux appels de fonction provenant du navigateur. Ce dernier peut être aussi appelé par le *plugin*. L'*API* de développement fournit l'ensemble de ces fonctions.

API de développement Il y a trois types de fonctions présentes dans la bibliothèque de développement de Netscape :

1. les fonctions de chargement et de libération du *plugin* (préfixe `NP_`);
2. les fonctions d'interface du *plugin* qui seront appelées par le navigateur (préfixe `NPP_`); à écrire par le programmeur;
3. les fonctions d'interface du navigateur qui seront appelées par le *plugin* (préfixe `NPN_`).

Lors de la détection d'une page contenant des données à traiter par un *plugin*, le navigateur effectue les actions de la figure 6 :

- il charge dynamiquement en mémoire le code du *plugin* (s'il ne l'est pas déjà) par la fonction `NP_Initialize`;
- il crée une nouvelle instance du *plugin* (fonction `NPP_New`) sachant que plusieurs instances peuvent être créées si nécessaire;
- il informe le *plugin* de l'identificateur de la fenêtre (`NPP_SetWindow`);
- il transmet les données aux *plugin* via un flux de communication et les événements survenant dans la fenêtre du *plugin*;
- quand la page de l'instance disparaît l'instance est détruite (fonction `NPP_Destroy`);
- quand la dernière instance d'un *plugin* est détruite alors le *plugin* est déchargé de la mémoire (fonction `NP_Shutdown`).

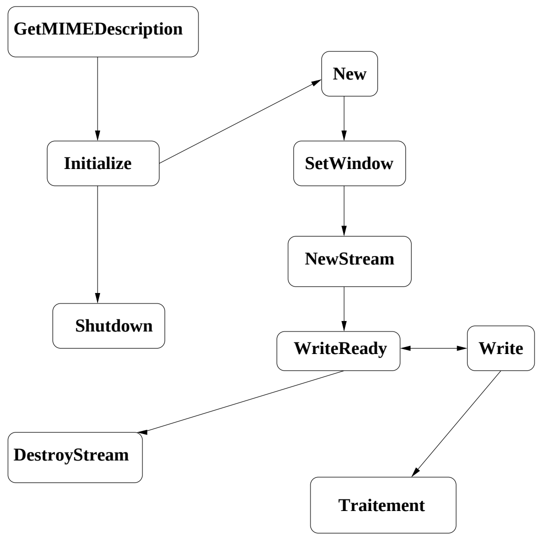

Figure 6: Schéma d'exécution d'un *plugin*

Le *plugin* peut alors gérer les événements de la fenêtre qui lui est associée par le navigateur en définissant ses propres fonctions de traitement d'événements et d'affichage.

2.2. Applications graphiques Objective Caml

Le *plugin* pour les applications graphiques Objective Caml suit la démarche précédente. Il est fourni sous forme d'une bibliothèque dynamique (`.so` pour Unix, `.dll` pour Windows). Lors de la demande de création de la première instance `NPP_New` l'identificateur de la fenêtre graphique fournie par le navigateur est conservée. A l'envoi des données dans le flux de communication un *toplevel*, construit sous forme d'une bibliothèque dynamique, est lancé. Celui-ci charge la bibliothèque `graphics`, met à jour la fenêtre graphique puis charge les données passées dans le *toplevel*. La bibliothèque `graphics` a été modifiée pour pouvoir forcer l'identificateur de la fenêtre (`set_window_id`).

```
  win = ((PluginInstance *) instance->pdata)->window;

  sprintf(headers,"Graphics.set_window_id %lu ;;",win);
  sprintf(src,"#use %s;;",fname);
  commandes[0]=headers;
  commandes[1]=src;
  pthread_create(&thread, NULL, (void*)toplevel_thread, (void*)commandes);

void * toplevel_thread(char* arg[2]) { ...
 (*toplevel_exec)(arg[0]);
 (*toplevel_exec)(arg[1]);  ...
}
```

Comme plusieurs instances peuvent être lancées et que la bibliothèque `graphics` ne peut adresser qu'une seule fenêtre graphique, il ne peut pas y avoir plusieurs programmes Objective Caml s'exécutant dans le même navigateur. Pour éviter de bloquer le navigateur par l'exécution du programme Objective Caml le lancement d'un programme est *threadé*. Néanmoins un seul *thread* est actif à un moment donné.

Les figures 7 et 8 montrent respectivement les applications graphiques `colorwheel`[12] dans le navigateur Mozilla sous X-Window et `deminer`[3] dans le navigteur FireBird sous Windows.

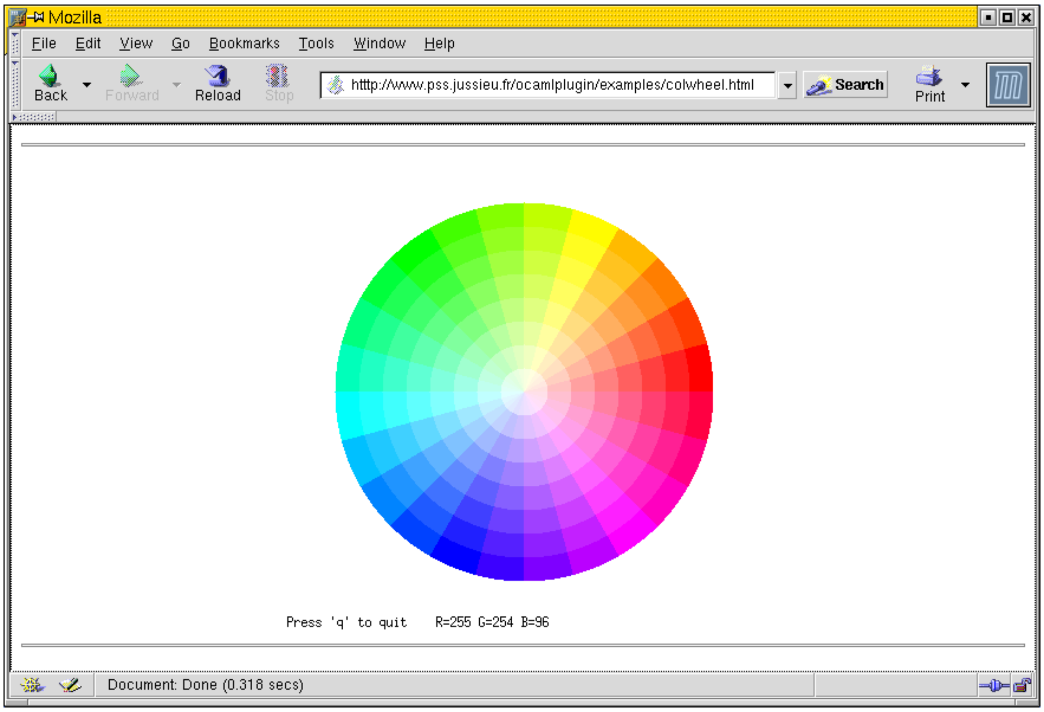

Figure 7: L'application Colorwheel sous Mozilla (X-window)

[12] `http://caml.inria.fr/Examples/eng.html`

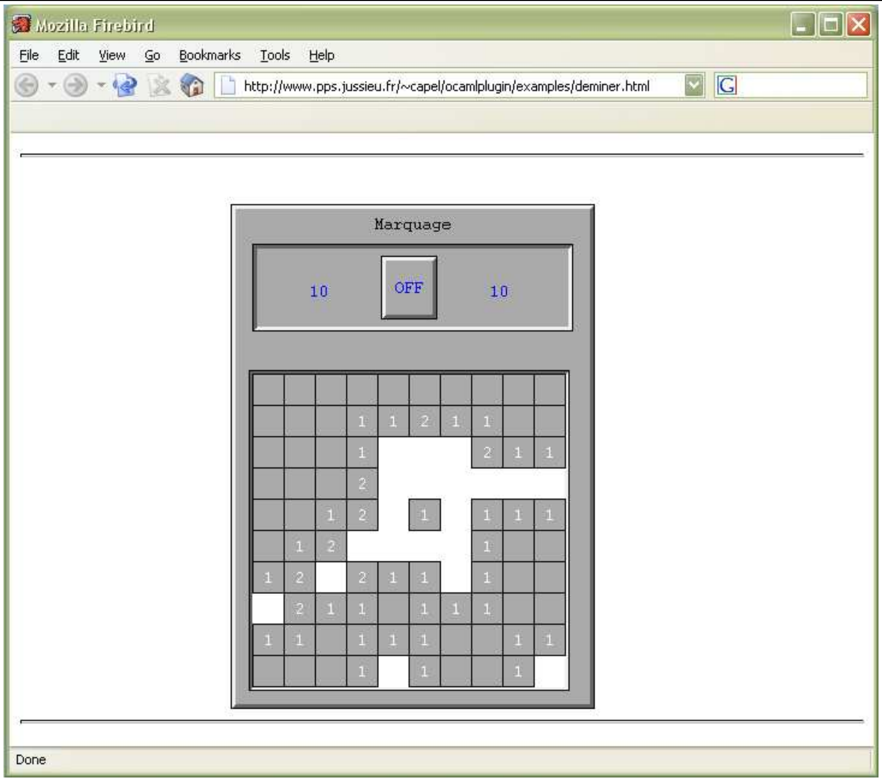

Figure 8: L'application Deminer sous FireBird (Windows)

La sortie de `Colorwheel` (appui de la touche 'q') efface la portion de fenêtre graphique du navigateur associée à l'application Objective Caml..

2.3. Choix d'implantation et limitations

Le choix de la bibliothèque `graphics` a été effectué pour sa portabilité. Cela oblige à n'avoir qu'une seule application graphique s'exécutant en même temps parce que cette bibliothèque ne sait gérer qu'une seule fenêtre. Ce *plugin* est en fait mono-*thread*.

Du point de vue sécurité des programmes téléchargés, la compilation des sources garantit que le code-octet exécuté est correct (dans la mesure où la bibliothèque *toplevel* l'est). Rien n'empêche par ailleurs de charger du code-octet. Les temps de transfert et compilation d'un texte source sont le plus souvent équivalents aux temps de transfert et de chargement du code-octet. Toutes les fonctions de la bibliothèque standard sont accessibles, y compris les lectures/écritures de fichiers. Une manière d'y remédier est de suivre les limitations de la `stdlib` comme présentées pour MMM.

L'application graphique doit être contenue dans un seul fichier source. Une évolution serait de pouvoir télécharger des fichiers d'un serveur *via* la primitive `use` en ayant accès à l'URL du serveur.

3. Excel et Objective Caml

Le tableur Excel [8] est un logiciel très répandu dans l'industrie, en particulier financière. De nombreux utilisateurs ne sont en effet pas des programmeurs professionnels au sens classique du terme mais sont néanmoins confrontés continuellement à la notion de programmation par l'intermédiaire de cette application.

Les versions récentes du tableur Excel peuvent être considérées comme la conjonction

- d'une interface utilisateur aisément manipulable et flexible, orientée vers le traitement de matrices de données. L'utilisateur manipule en effet des «cellules» qui sont organisées en lignes et colonnes. Chaque case peut contenir une donnée élémentaire (nombre, date, chaîne de caractères, booléen,...) et une éventuelle formule permettant de calculer cette valeur à partir d'autres cases et de fonctions standards.
- et d'un environnement de programmation Visual Basic, langage de programmation de Microsoft réputé simple et accessible.

Souvent, seul le premier aspect est utilisée. Remarquons que, dans ce cas, l'utilisateur écrit des relations fonctionnelles entre cellules (le logiciel «exécute» ce calcul en analysant le graphe de dépendance): il réalise en fait un programme dans un langage fonctionnel simple (sans effets de bord), non typé (l'addition d'une chaîne de caractère et d'un nombre renvoie une valeur spécifique, `#ERR`), et plat (pas de définitions locales de fonctions).

Cette fonctionnalité peut être étendue avec des fonctions écrites en Visual Basic. De telles fonctions peuvent alors être utilisées comme les fonctions standards dans les formules contenues dans les cellules. Leur résultat doit être une valeur de type simple pouvant être affichée dans une cellule [13]; dans le cas contraire, une erreur est générée à l'exécution. De telles fonctions Visual Basic peuvent lancer des programmes externes ou appeler des bibliothèques dynamiques. Cette approche est utilisée de manière abondante dans l'industrie et il paraît donc naturel d'envisager l'intégration dans Excel d'un système Objective Caml complet.

3.1. Communication Excel, Visual Basic, COM

L'absence de typage est le résultat de l'utilisation fréquente, dans le langage Visual Basic, du type `variant` de Microsoft. La notion de `variant` est essentiellement une valeur de type dynamique, composée d'un octet identifiant le type et d'une zone «valeur» contenant des octets interprétés selon le type (type union de C). Cette convention n'est pas propre à Visual Basic, mais est un type défini dans l'architecture d'interopérabilité binaire COM, promue par Microsoft.

Remarquons qu'un tel variant peut contenir un ensemble pré-défini de valeurs simples [14], des valeurs spéciales d'erreur, d'absence de résultat (`NUL`) ainsi que diverses formes de tableaux dont des tableaux de `variant`s, ce qui permet de décrire, certes avec lourdeur, des structures arborescentes [15].

De nombreuses bibliothèques et *add-ons* de Excel proposent ainsi des fonctions renvoyant des `variant`s que le programmeur peut alors déstructurer pour traitements et analyses.

3.2. Macros et calcul en Objective Caml

La technologie du *toplevel* embarqué se prête tout naturellement à être utilisée dans un tel environnement, afin de donner la possibilité de réaliser aisément sous Excel des «calculs» en Objective

[13] ou, dans certains cas, une matrice de valeurs de types simples qui pourra être placée dans un ensemble rectangulaire de cellules.

[14] on retrouve les divers types du langage C, ainsi que diverses formes de dates et de temps.

[15] bien sûr, ni une fermeture, ni même un quelconque pointeur de fonction ne peut être représenté dans un `variant`.

Caml. Une solution simple consiste à *générer*, puis exécuter des programmes Objective Caml en retournant les résultats sous forme de valeurs `variant`. Une fonction Visual Basic pourra donc générer un programme en fonction de certains paramètres, que le toplevel embarqué va compiler et exécuter pour permettre à la fonction Visual Basic de récupérer ce résultat. La possibilité de générer de tels programmes *spécialisés* (évaluation partielle) à l'exécution en fonction de paramètres connus uniquement à l'exécution peut, dans certains cas, permettre des gains en efficacité importants. Pour de nombreux calculs financiers, ceci est effectivement le cas [4].

Bien entendu, les `variants` n'ont pas la richesse d'expression des types de Objective Caml: on ne peut distinguer les vecteurs et les listes, il est impossible de représenter des constructeurs ou des valeurs fonctionnelles... Nous implantons donc une *projection* des valeurs Objective Caml vers les `variants`. Cette heuristique rudimentaire est toutefois utilisable en pratique. La figure 9 donne une syntaxe abstraite des valeurs `variant` que nous supportons.

```
(* AST for the (subset of the) variant values we support. *)
type variant =
  | VT_empty
  | VT_bool of bool
  | VT_int of int
  | VT_float of float
  | VT_string of string
  | VT_array1 of variant array (* one dim variant array *)
  | VT_array2 of variant array * int * int (* two dim variant array with size *)
```

Figure 9: représentation du type `variant` en Objective Caml

La fonction de projection transforme une valeur de type `out_value` de Objective Caml[16] en une valeur de type `variant` [17].

Nous projetons (voir figure 10), entre autres:

- les types de base (entiers, flottants,...) vers leurs représentations évidentes
- les listes, les tableaux et les n-uplets vers des tableaux de variants à une dimension
- les enregistrements vers des tableaux de variants à une dimension, en éliminant les noms des champs
- les constructeurs sans arguments vers des chaînes de caractères
- le constructeur `None` vers la valeur `Empty`, alors que `Some v` est traité comme `v`
- les constructeurs avec arguments vers des tableaux de variants, le premier élément étant une chaîne de caractère

Remarquons que la fonction `variant` de la figure 10 peut être assimilée à un imprimeur particulier de valeurs du *toplevel*.

3.3. Résultat de type matrice

Excel traite de manière particulière les résultats de type matrice de `variants`. Un tel résultat peut être aisément placé dans un *ensemble rectangulaire* de cellules. On peut donc souhaiter pouvoir aisément

[16] `out_value` décrit la syntaxe abstraite des valeurs renvoyées par le toplevel et est défini dans le module `Outcometree`.
[17] une telle valeur est ensuite encore mise au format binaire COM exigé par Microsoft, mais ceci ne présente aucun intérêt particulier ici.

```
(* projection of OCaml values onto the variant type. *)
let rec variant =
  let ident = function
    | Oide_apply _ -> "unknwon"
    | Oide_dot (_, s) -> s
    | Oide_ident s -> s in
  let map l = List.map variant l in
  let array1 tl = VT_array1(Array.of_list tl) in
  function
    | Oval_int i -> VT_int i
    | Oval_float f -> VT_float f
    | Oval_char c -> VT_string (Char.escaped c)
    | Oval_string s -> VT_string s
    | Oval_list tl
    | Oval_array tl
    | Oval_tuple tl -> array1 (map tl)
    | Oval_record tl ->
        let tl = List.map (fun (_, x) -> variant x) tl in
        array1 (tl)
    | Oval_constr (Oide_ident "true", []) -> VT_bool true
    | Oval_constr (Oide_ident "false", []) -> VT_bool false
    | Oval_constr (Oide_ident "None", []) -> VT_empty
    | Oval_constr (Oide_ident "Some", [t]) -> variant t
    | Oval_constr (Oide_ident "ExcelMatrix", [t]) -> matrix(variant t)
    | Oval_constr (name, []) -> VT_string (ident name)
    | Oval_constr (name, tl) ->
        array1 ((VT_string (ident name)) :: map tl)
    | Oval_variant (name, _) -> VT_string name
    | Oval_stuff s -> VT_string s
    | Oval_ellipsis -> VT_string "..."
    | Oval_printer _ -> assert false (* treated elsewhere. *)
```

Figure 10: projection des valeurs Objective Caml vers un `variant`

transformer toute valeur renvoyée par le toplevel en une telle matrice. Afin de simplifier le travail du programmeur, nous avons défini dans `pervasives.ml` un constructeur polymorphe, `ExcelMatrix`, qui est détecté dans la routine de projection (voir figure 10) pour coerciser le résultat vers une valeur de type matrice de `variants`.

constructeur de type «tableau Excel»

```
(** 2 Excel and COM interface *)
type 'a ms_com = ExcelMatrix of 'a
```

Il est en effet fréquent de vouloir représenter sous forme tabulaire un résultat Objective Caml qui n'est pas, a priori, une «matrice». Une liste d'enregistrements ou de n-uplets est ainsi un exemple typique de cette situation. Il devient alors aisé d'utiliser Excel comme simple front-end graphique d'affichage de résultats tabulaires obtenus dans Objective Caml, ce que montre l'exemple suivant.

3.4. Exemple

Nous donnons un exemple trivial : une fonction Visual Basic engendre un programme Objective Caml de calcul paramétré (figure 11). Les fonctions `com_toplevel_init` et `com_toplevel_exec` correspondent aux fonctions définies dans la bibliothèque *libtoplevel.dll*. La fonction `ocaml` initialise si nécessaire le *toplevel* et exécute le programme Objective Caml passé en argument.

La fonction `ocaml_calculation` est appelée sur la ligne des formules avec les cellules de la ligne 2 de la feuille de calcul (B2,C2,D2,E2,F2,G2).

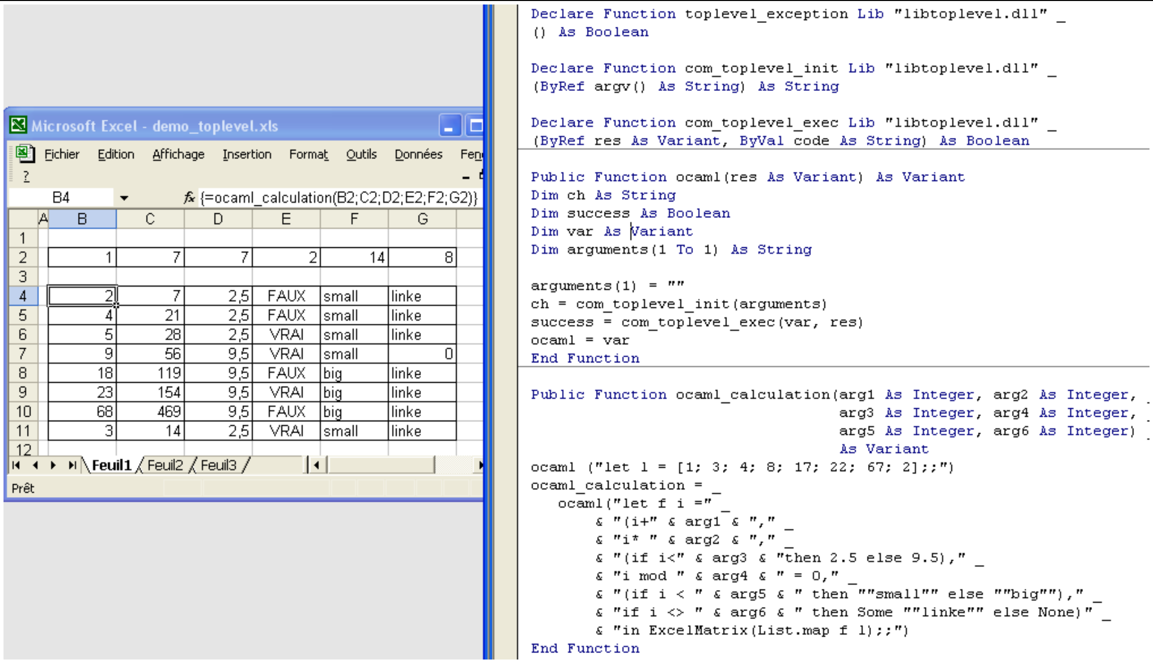

Figure 11: macro `VBA` + Objective Caml

Elle envoie alors au *toplevel* le programme suivant.

```
let l = [1; 3; 4; 8; 17; 22; 67; 2];;
let f i =
  i+1,
  i*3,
  if i < 11 then 2.5 else 9.5,
  i mod 2 = 0,
  if i < 14 then "small" else "big",
  if i <> 8 then Some "linke" else None
in
  ExcelMatrix (List.map f l);;
```

Remarquons que nous renvoyons un résultat «annoté» avec `ExcelMatrix`, qui sera donc une matrice pour Excel qui affichera ce résultat (une liste de huit sextuplets) sous la forme d'un tableau (B4-G11). Chaque ligne correspond à un élément de la liste.

Conclusion

Le besoin d'embarquer le compilateur Objective Caml s'est plusieurs fois exprimé dans la liste de diffusion de Caml[18] pour différents types d'applications. Cet article a proposé une solution technique[19] pour utiliser le *toplevel* Objective Caml sous forme d'une bibliothèque et a montré son utilisation comme extensions pour des logiciels grand public.

[18] `http://caml.inria.fr/caml-list-eng.html`
[19] `http://www.pps.jussieu.fr/~capel`

L'exécution d'une application graphique dans la fenêtre d'un navigateur autorise une meilleure diffusion de programmes Objective Caml. Ce *plugin* n'aborde pas les problèmes de sécurité (lecture et écriture sur le disque, ...). Il se rapproche plus du *plugin* de Common Lisp[20] que des *applets* `Java`. Il lui manque surtout une bibliothèque graphique gérant plusieurs fenêtres.

L'intérêt de la compilation à la volée est de pouvoir charger dynamiquement du code non connu au démarrage de l'application, en particulier quand ce texte source est engendré lui-même par l'application. Cela autorise de construire des programmes spécialisés par évaluation partielle. L'interaction avec Excel montre une telle génération dynamique. Elle est en cours d'intégration dans un produit industriel de la société LexiFi[21].

Cette compilation dynamique de source est aussi utilisable pour les compilateurs dont le langage cible est Objective Caml ce qui permet alors d'embarquer ces langages. L'utilisation de l'outil `camlp4`[22] facilite ce type d'implantation dans la mesure où l'on peut charger dynamiquement au niveau de la bibliothèque *toplevel* des grammaires, et même les étendre dynamiquement si besoin est.

Nous espérons que d'autres applications utiliseront ce *toplevel* embarqué. On pense tout particulièrement aux cédéroms d'auto-formation à la programmation[2] utilisant `Scheme` développés à l'université Pierre et Marie Curie. Ils nécessitent un confinement dans l'exécution des programmes des étudiants. Une partie de cette exécution limitée rejoint la sécurité des *plugins web*. La détection de programmes qui bouclent peut être réalisée par transformation de programmes en ajoutant des points de contrôle dans les soumissions des étudiants.

Bibliographie

[1] Bosser, A.-G., and Alberti, F. L'expérience Scol. In *Journées Francophones des Langages Applicatifs (JFLA2003)* (Jan. 2003), INRIA.

[2] Brygoo, A., Durand, T., and et al., P. M. Un cédérom pour scheme, chacun son entraîneur, un entraîneur pour tous. In *Colloque Internationnal sur les T.I.C.E. d'ingénieur et de l'industrie* (Nov. 2002).

[3] Chailloux, E., Manoury, P., and Pagano, B. *Développement d'Applications avec Objective Caml*, 1st ed. O'Reilly, 2000. on-line english version : `http://caml.inria.fr`.

[4] Eber, J.-M., and Peyton-Jones, S. Composing contracts: an adventure in financial engineering. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)* (2000), pp. 280–292. `http://citeseer.nj.nec.com/jones00composing.html`.

[5] Leroy, X. The objective caml system : documentation and user's manual. Tech. rep., INRIA, 2002. `http://caml.inria.fr`.

[6] Netscape. *Netscape Gecko Plug-in API Reference*, 2002. `http://devedge.netscape.com/library/manuals/2002/plugin/1.0/`.

[7] Rouaix, F. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking* (May 1996), vol. 28, Elsevier, pp. 1365–1371.

[8] Steven Roman. *Writing Excel Macros with VBA*, 2nd ed. O'Reilly, 2002.

[9] Taguet, A., Pejcic, N., and Zinovieff, N. Caml applet support for netscape, elaboration report, 2002. université Pierre et Marie Curie, TER de maîtrise d'informatique.

[20] `http://www.franz.com/support/documentation/6.1/doc/plugin.htm`

[21] `http://www.lexifi.com`

[22] `http://caml.inria.fr/camlp4`