

Observation temps-réel de programmes Caml

S. Conchon^{1,2,3} & J.-C. Filliâtre^{2,1,3} & F. Le Fessant³ & J. Robert¹ & G. Von Tokarski¹

1: Université Paris Sud F-91405 Orsay

2: CNRS / LRI UMR 8623 F-91405 Orsay

3: INRIA Saclay – Île-de-France F-91893 Orsay

{conchon,filliatr,jrobert,gvt}@lri.fr, fabrice.le_fessant@inria.fr

Résumé

Pour mettre au point un programme, tant du point de vue de sa correction que de ses performances, il est naturel de chercher à observer son exécution. On peut ainsi chercher à observer la gestion de la mémoire, le temps passé dans une certaine partie du code, ou encore certaines valeurs calculées par le programme. De nombreux outils permettent de telles observations (moniteur système, *profiler* ou *debugger* génériques ou spécifiques au langage, instrumentation explicite du code, etc.). Ces outils ne proposent cependant que des analyses « après coup » ou des observations très limitées. Cet article présente Ocamlviz, une bibliothèque pour instrumenter du code OCaml et des outils pour visualiser ensuite son exécution, en temps-réel et de manière distante.

1. Introduction

[1]

OCaml let f x = x+1

2. Principe

Ocamlviz est à la fois une bibliothèque pour instrumenter du code OCaml à observer et des outils pour transmettre et visualiser les résultats. Le principe de fonctionnement est illustré figure ?? . En premier lieu, l'utilisateur instrumente son code en indiquant les observations qu'il souhaite réaliser. Il utilise pour cela les briques de base fournies par la bibliothèque Ocamlviz. Il lie ensuite son programme avec cette bibliothèque. Le code ainsi obtenu peut être exécuté normalement mais il peut également être observé. En effet, en parallèle de l'exécution normale, le programme se comporte maintenant comme un serveur qui attend des connections et transmet le résultat des observations à ses clients. En particulier, Ocamlviz fournit une interface graphique évoluée permettant une visualisation agréable et synthétique des résultats (voir figure 2).

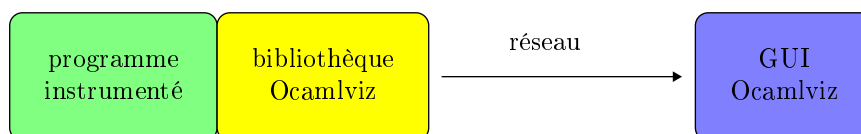


FIG. 1 – Architecture d'Ocamlviz.

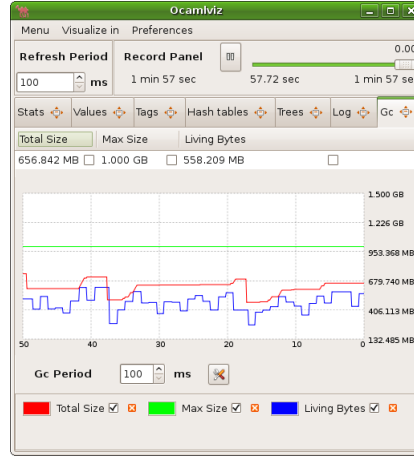


FIG. 2 – L’interface graphique d’Ocamlviz.

Le caractère client/serveur de l’architecture offre de nombreux avantages. Premièrement, il permet de décider d’observer le programme à n’importe quel moment de son exécution. En particulier, un client peut se connecter longtemps après le début de l’exécution et se déconnecter à tout instant. Deuxièmement, un nombre arbitraire de clients peuvent se connecter à un même programme en cours d’exécution. Troisièmement, les clients peuvent se connecter depuis des machines distantes, qu’elles aient ou non la même architecture que celle sur laquelle s’exécute le code instrumenté. Enfin, cette architecture dissocie l’instrumentation de l’observation des résultats. En particulier, le protocole de communication est indépendant du langage OCaml et permet par exemple l’utilisation de clients écrits dans d’autres langages.

L’instrumentation minimale consiste simplement à lier son programme avec la bibliothèque Ocamlviz. Cela a pour effet immédiat d’envoyer aux clients des données relatives au GC d’OCaml (taille totale du tas et taille de sa partie vivante). L’interface graphique d’Ocamlviz se charge alors d’afficher cette information sous forme d’un graphe, tel qu’on peut le voir sur la figure 2. Au delà de cette première information, la bibliothèque Ocamlviz offre les possibilités suivantes :

- mesurer le temps passé entre deux points du programme ;
- compter le nombre de passages en un ou plusieurs points du programme ;
- observer des valeurs calculées par le programme ;
- mesurer le nombre et la taille totale d’un ensemble de valeurs désignées par l’utilisateur ;
- afficher des messages à la `printf` qui sont archivés dans un journal.

Le reste de cette section détaille ces différentes instrumentations.

Mesure du temps. Ocamlviz fournit une notion de chronomètres, que l’on peut déclencher et arrêter en tout point du programme. Ces points de programme ne coïncident pas nécessairement avec le début et la fin d’une fonction, comme c’est le cas généralement dans les outils de *profiling*. D’autre part, on peut mesurer, par accumulation, le temps passé dans plusieurs sections du programme. On crée un chronomètre de la manière suivante : `OCaml let chrono = Ocamlviz.Time.create "t"` La chaîne de caractères "t" n’est utile que pour l’affichage dans l’interface graphique. Le chronomètre s’utilise alors ainsi : `OCaml let f x = ... Ocamlviz.Time.start chrono ; let z = ... in Ocamlviz.Time.stop chrono ...` Dans cet exemple, on mesure le temps passé dans le calcul de `z` mais pas dans le reste de la fonction `f`. Il est important de noter que le chronomètre peut être déclenché dans une fonction et arrêté dans une autre. La figure 3 montre comment l’interface graphique présente la valeur du chronomètre "t" (ici 49,9 secondes représentant 100% du temps total d’exécution).

Stats	Values	Tags	Hash tables	Trees	Log	Gc
Name	Count	Time				
p	803800206	40.84 sec	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
t		<input type="checkbox"/> 40.946559	40.94 sec	<input type="checkbox"/> 100. %	<input type="checkbox"/>	<input type="checkbox"/>

FIG. 3 – Chronomètres et points de programme.

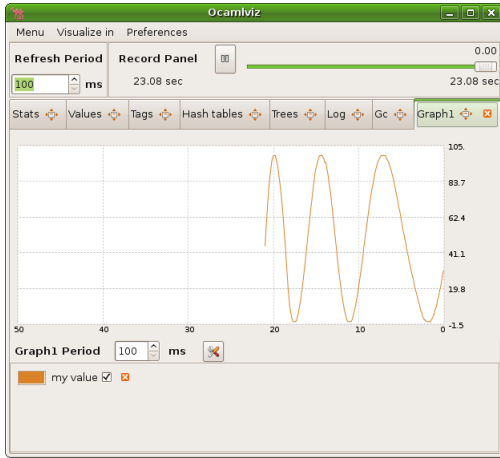


FIG. 4 – Observation de valeur scalaire.

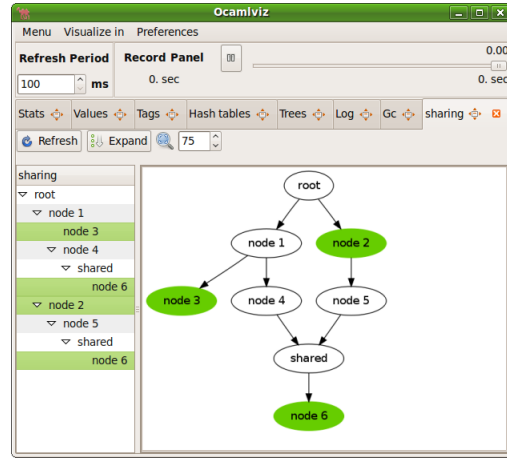


FIG. 5 – Observation d'une valeur structurée.

Points de programme. Ocamlviz fournit un moyen de marquer un ou plusieurs points de programme puis de compter le nombre de fois que l'exécution passe par ces marqueurs. On crée un tel marqueur de la manière suivante : `OCaml let point = Ocamlviz.Point.create "p"` On marque alors les points de programme que l'on souhaite observer avec `OCaml ... Ocamlviz.Point.observe point ; ...` Un même marqueur peut être utilisé à plusieurs endroits du programme et le décompte est global à chaque marqueur. La figure 3 montre ainsi que l'on est passé 803 800 206 fois par des points de programme marqués avec `point`.

Observer des valeurs. Ocamlviz permet d'observer des valeurs arbitraires calculées par le programme. Pour les types simples (entiers, flottants ou chaînes de caractères), il suffit de fournir une fonction calculant la valeur à observer. Dès lors, cette fonction est évaluée régulièrement, à une fréquence que l'utilisateur peut spécifier. Par exemple, le code suivant `OCaml let () = Ocamlviz.Value.observe_float_fct "myvalue" period : 200 (fun () -> sin !v)` déclare une fonction d'observation calculant la valeur flottante `sin !v` toutes les 200 millisecondes. La figure 4 montre la visualisation de cette valeur dans l'interface graphique.

Pour les types plus complexes, une solution simple consiste à transformer la valeur à observer en chaîne de caractères. Cependant, Ocamlviz fournit un moyen plus élégant d'observer des valeurs telles que des arbres ou des graphes. Pour cela, l'utilisateur commence par transformer sa valeur dans un type de la forme suivante : `OCaml type t = node : string; mutable children : t list` Il s'agit donc, en toute généralité, d'un type de graphes dont les nœuds sont étiquetés par des chaînes de caractères. L'aspect `mutable` du champ `children` permet en effet de construire des valeurs cycliques. Si la donnée de type `t` construite par l'utilisateur contient du partage, celui-ci sera préservé par le protocole de communication et présenté dans l'interface graphique fournie par Ocamlviz. La figure 5 illustre la

visualisation d'une valeur structurée où le nœud étiqueté "**shared**" est partagé.

Marquer des valeurs. Ocamlviz permet d'analyser l'utilisation de la mémoire plus finement qu'à travers les informations globales fournies par le GC. Ocamlviz donne en effet la possibilité de marquer des valeurs puis de connaître, à tout instant, le nombre de ces valeurs toujours vivantes et l'espace mémoire qu'elles occupent.

Comme pour les points de programme, on commence par créer un marqueur : OCaml `let t = Ocamlviz.Tag.create size:true period:300 "foo"`. On crée ici un marqueur de nom "**foo**". On spécifie que l'on souhaite calculer la taille occupée (option **size**) et la calculer toutes les 300 millisecondes (option **period**). On peut par exemple marquer une valeur qui vient d'être construite, comme dans la fonction suivante : OCaml `let cons x = let l = Random.float 10. : x in Ocamlviz.Tag.mark t l; l`. L'interface graphique permet de visualiser en temps réel le nombre et la taille de chaque marqueur.

Name	Count	Max Count	Size	Max Size	Overall Size
foo	6032 6.900 sec	6032 6.900 sec	289.560 KB 6.900 sec	289.560 KB 6.900 sec	0.03 %

Sur cette capture, on lit que 6 032 valeurs encore vivantes sont marquées avec le tag "**foo**" et qu'elles occupent un peu plus de 289 ko.

Un même marqueur peut être utilisé pour marquer une ou plusieurs valeurs, qu'elles soient ou non du même type. Si plusieurs données sont marquées avec le même marqueur, et qu'elles partagent des valeurs, alors les données partagées ne sont comptées qu'une seule fois dans le calcul de l'espace mémoire occupé. Ainsi dans le code suivant OCaml `let l2 = [Random.int 3; Random.int 4] in Ocamlviz.Tag.mark t l2; let l3 = Random.int 5 : : l2 in Ocamlviz.Tag.mark t l3; ...` le nombre de valeurs marquées par le tag **t** est 2 (les listes **l2** et **l3** et la taille mémoire occupée pour ce tag est de 36 octets (3 blocs *cons*, de 3 mots chacun en comptant l'entête de bloc).

Journal. Enfin, Ocamlviz fournit une facilité « à la **printf** »

3. Réalisation

Comme illustré sur la Figure 6, Ocamlviz se décompose en une bibliothèque serveur liée au programme instrumenté, qui communique via le réseau avec un ou plusieurs clients, et une bibliothèque qui permet d'écrire aisément des clients.

Dans cette section, nous décrivons comment le serveur et les clients ont été implantés dans la version courante d'Ocamlviz.

3.1. Protocole de communication

Une capacité importante d'Ocamlviz est la possibilité d'observer le fonctionnement d'un programme s'exécutant sur une machine depuis une autre machine. Pour permettre une hétérogénéité maximale entre ces deux machines, un protocole binaire et portable a été spécifié et implanté dans les bibliothèques serveur et client. Ceci permet d'une part la communication entre des machines d'architectures différentes (32 et 64 bits par exemple, mais aussi *little-endian* et *big-endian*¹, mais aussi l'écriture de clients et de serveurs dans d'autres langages qu'Objective-Caml. Ainsi, il sera possible dans

¹Les termes anglais *little-endian* et *big-endian* ont été empruntés aux *Voyages de Gulliver* de Jonathan Swift. Il serait donc naturel de les traduire en français par « petits-boutien » et « gros-boutien ».

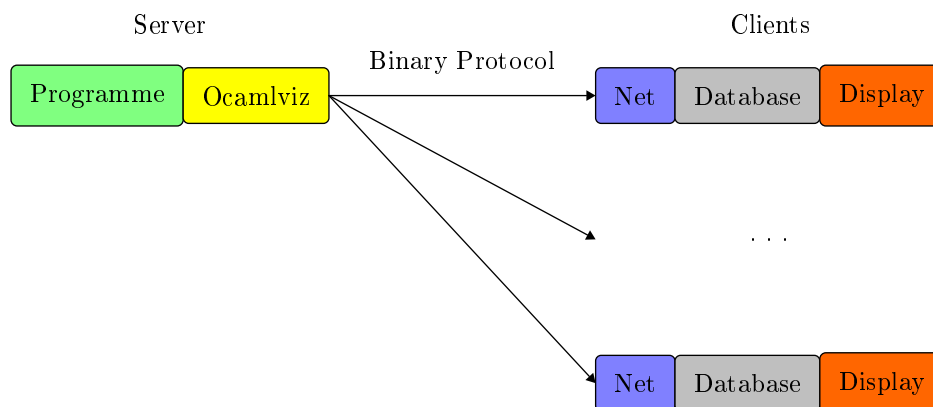


FIG. 6 – Ocamlviz est composé d’une bibliothèque liée au programme instrumenté, qui communique via le réseau avec un ou plusieurs clients.

l’avenir d’écrire des bibliothèques serveurs permettant d’instrumenter des programmes dans d’autres langages qu’Objective-Caml et de les observer depuis le client graphique actuel d’Ocamlviz, mais aussi d’écrire de nouveaux clients dans d’autres langages.

3.1.1. Messages

Le protocole actuel ne contient que des messages du serveur vers les clients. Il se compose de trois messages :

OCaml type msg = | Declare of uid * kind * string | Send of uid * value | Bind of uid list

Les messages désignent des valeurs observées, qui sont identifiées par des entiers uniques de type uid. Le message **Declare** (uid, kind, name) déclare au client une nouvelle valeur observée, en indiquant son identifiant, sa nature de type kind et le nom qui sera utilisé pour l’affichage. Au moment de sa connexion, le client reçoit du serveur un ensemble de messages **Declare** correspondant à toutes les valeurs observées dans le programme instrumenté jusqu’à cet instant.

Le message **Send** (uid, v) fournit une mise à jour de la valeur courante de l’identifiant uid avec la valeur v. Il est envoyé régulièrement pour chaque valeur, même si celle-ci n’a pas changé entre-temps.

Enfin, le message **Bind** uid_list indique au client qu’un certain nombre d’identifiants sont liés entre eux ; c’est en particulier le cas pour les deux valeurs correspondant au nombre et la taille d’un marqueur de type Ocamlviz.Tag.t.

3.1.2. Protocole binaire

Chaque message est une chaîne de caractères : elle se décompose d’un premier entête de 4 octets, indiquant la longueur totale de la chaîne, d’un second entête indiquant le type du message, puis enfin des arguments du message, dont l’ordre et le type dépendent du type du message.

Cette représentation permet d’effectuer une analyse efficace de chaque message. La connaissance de sa longueur dès l’entête permet notamment de ne commencer l’analyse que lorsque tout le message a été lu. Elle permet aussi de n’allouer une chaîne de caractères pour la lecture du message que si la taille de celui-ci dépasse la taille de la chaîne par défaut (65000 actuellement), évitant les allocations de petites chaînes de caractères qui fragmentent le tas et ralentissent le fonctionnement du ramasse-miettes.

Pour faciliter l'écriture et l'extension du protocole, des fonctions sont définies pour transmettre chaque type Objective-Caml de base, puis combinées pour transmettre les types plus complexes. Pour chaque type de base *ttype* (entiers 8, 16, 32 ou 64 bits, flottant, etc.) on introduit le couple de fonctions suivant :

```
val get_ttype : string → int → ttype × int
val buf_ttype : Buffer.t → ttype → unit
```

La fonction `get_ttype s pos` extrait une valeur de type *ttype* de la chaîne *s* à partir de la position *pos* et renvoie un couple contenant cette valeur et la position suivante dans la chaîne. La fonction `buf_ttype buf v` encode la valeur *v* de type *ttype* à la fin du tampon *buf*.

Pour insérer la taille de chaque message, quatre octets nuls sont placés en tête du tampon avant l'encodage du message. La chaîne correspondant au message est ensuite extraite du tampon puis, sa longueur étant connue, ses quatre premiers octets sont modifiés en conséquence.

Enfin, pour permettre une compatibilité entre architectures 32 et 64 bits, les valeurs manipulées par le client portent une marque de type, en particulier indiquant pour les entiers s'ils sont sur 32 (pour les `int` 31 bits et les `int32`) ou 64 bits (pour les `int` 63 bits et les `int64`).

3.2. Bibliothèque serveur

Le bibliothèque serveur calcule toutes les 100 millisecondes l'ensemble des valeurs en cours d'observation et les envoie à tous les clients connectés. Elle gère aussi les connexions de nouveaux clients. Le délai entre les observations peut être modifié en utilisant une variable d'environnement `OCAMLVIZ_PERIOD`.

Deux mécanismes — les alarmes et les processus légers (*threads*) — ont été implantés pour effectuer ces opérations régulières, suivant les contraintes liées au programme à observer.

3.2.1. Les alarmes

Les alarmes permettent de déclencher l'exécution d'une fonction à intervalles de temps réguliers. Cette technique fonctionne bien dans la plupart des cas, car elle interrompt l'exécution du programme complètement et ne souffre donc pas de problème de synchronisation. Il existe néanmoins certains cas où les alarmes peuvent poser problème :

- Les alarmes ne peuvent pas interrompre le programme à n'importe quel instant. En effet, Objective-Caml ne permet l'exécution du code associé à une alarme qu'à certains moments particuliers, afin d'éviter les interactions avec le ramasse-miettes. En particulier, Caml ne permet la gestion des alarmes qu'au moment des allocations et des entrées-sorties. Aussi, un programme qui ne fait que calculer, sans allocation ni entrée-sortie, ne verra jamais ses alarmes traitées et le serveur `Ocamlviz` n'enverra aucune donnée. Pour remédier à ce problème, il est possible d'insérer dans le code un appel `Ocamlviz.yield ()` qui donne l'opportunité à Caml de traiter les alarmes, le cas échéant.
- Quand le programme utilise déjà des alarmes, il devient impossible pour `Ocamlviz` de les utiliser, car il n'y qu'un seul gestionnaire associé aux alarmes. Il devient alors nécessaire d'utiliser les processus légers pour ne pas modifier la sémantique du programme d'origine.

3.2.2. Les processus légers

Cette deuxième solution consiste à lancer au démarrage du programme un processus léger, dont la seule tâche est d'appeler régulièrement la fonction d'observation des valeurs. Cependant, cette solution comporte également des inconvénients :

- L'utilisation de processus légers a certaines limites en Objective-Caml, en particulier parce que le ramasse-miettes n'est pas concurrent. En conséquence, la plateforme d'exécution ne permet l'exécution de code Objective-Caml que d'un seul processus léger à la fois.
- Comme pour les alarmes, l'ordonnancement des processus légers ne s'effectue pas à n'importe quel instant, mais uniquement lors des allocations et des entrées-sorties. Là encore, un appel explicite à `Ocamlviz.yield ()` peut être nécessaire pour permettre au processus léger d'observation de s'exécuter un court instant.

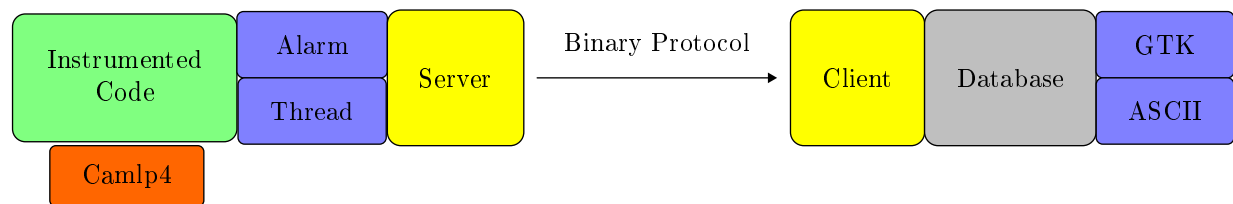
3.3. Observations

weak pointers, calcul de taille, GC

attention : données statiques allouées une seule fois / tout est compté, en profondeur / fonctionne correctement avec les données cycliques, polymorphes, etc.

3.4. Client

structures de données BD



4. Conclusion et perspectives

Remerciements. TODO : Jane Street

Références

- [1] Le langage Objective Caml. <http://caml.inria.fr/>.

