frama C
Software Analyzers

Comment un chameau peut-il écrire un journal ?

Julien Signoles

DE LA RECHERCHE À L'INDUSTRIE
cea list

avec la participation amicale de Sylvain Conchon
dans le rôle de Benjamin Monate

JFLA'14 9 janvier 2014

Contexte

- Frama-C est une plateforme d'analyse de code C
- développée en OCaml
- utilisée notamment pour obtenir des garanties fortes sur des programmes embarqués critiques
- successeur des outils Caveat (CEA) et Caduceus (Inria)

http://frama-c.com

Formalisation des besoins

Il était une fois...

Un jour de 2008,

Benjamin Monate vient me voir pour m'exprimer

un nouveau besoin pour Frama-C.

En bons défenseurs des spécifications formelles que nous sommes,

nous eûmes la conversation suivante...

Aperçu de la discussion

Le *quoi*

BM [1] Çà serait bien si on avait un mécanisme de journal à la Caveat dans Frama-C. T'aurais une idée ?

JS Euh, c'est quoi le journal de Caveat ?

BM C'est un fichier OCaml généré par Caveat, qu'on peut charger dynamiquement et qui reproduit les actions utilisateurs effectuées lors d'une session précédente.

1. Discussion reproduite avec son accord

Aperçu de la discussion

Le *pourquoi*

JS C'est rigolo, çà sert à quoi ?

BM Pour la certification DO-178B de Caveat par Airbus, il faut être en mesure de reproduire les résultats. Çà remplit ce rôle.

BM En plus, çà facilite les rapports de bogues par les utilisateurs qui n'ont qu'à envoyer leurs journaux.

BM Airbus s'en sert aussi pour automatiser certaines actions répétitives dans la GUI.

Aperçu de la discussion

Le *comment*

JS Et çà marche comment ?

BM Ben, je sais pas très bien. En tout cas, c'est en C++ et très *unsafe*, avec plein de casts, mais ce serait cool de le faire en OCaml dans Frama-C.

JS Bon, j'y réfléchis, mais çà me paraît difficile à faire, surtout sans `Obj.magic`...

BM Ce serait toujours mieux qu'en C.

Formalisation des besoins

Exemple

Après la formalisation,

un exemple...

frama C
Software Analyzers

Principe de la journalisation

`val register:` $(\alpha \to \beta) \to \alpha \to \beta$

- le développeur enregistre chaque fonction f à journaliser de façon à
- modifier automatiquement f pour que toutes ses applications totales soient écrites dans le journal

Procédures sans argument

Comment journaliser `f: unit → unit` ?

- imprimer "f ();" dans le journal

```
let print_sentence name fmt =
  Format.fprintf fmt "%s ();" name

(* val register:
     string → (unit → unit) → unit → unit *)
let register name f () =
  Sentences.add (print_sentence name);
  f ()

(* exemple issu du module File: *)
let init_from_cmdline =
  register "File.init_from_cmdline" init_from_cmdline
```

Fonctions mono-argument

Comment journaliser `f:` $\tau \rightarrow$ `unit` ?

- nécessite d'imprimer une valeur OCaml sous forme de code
- fonctionalité fournie par la bibliothèque de typage dynamique de Frama-C [Signoles@JFLA'11]

```
val pretty_code (* in module Datatype *):
  α Type.t → Format.formatter → α → unit
```

- prendre une valeur de type α `Type.t` en argument

```
val register:
  string → α Type.t → (α → unit) → α → unit
```

Fonctions mono-argument

Code

```
let print_sentence name ty_x x fmt =
  Format.fprintf fmt "%s %a;"
    name
    (Datatype.pretty_code ty_x) x

let register name ty_x f x =
  Sentences.add (print_sentence name ty_x x);
  f x
```

Appels de fonctions elle-même journalisées

Soit `f` et `g` deux fonctions journalisées telles que `f` appelle `g`.

- lors d'un appel à `f` avec un argument `x`
- `f x` est écrit dans le journal
- puis `f x` est réellement exécuté
- engendrant un appel à `g` avec un argument `y`
- qui génère l'écriture de `g y` dans le journal avant son exécution
- au final `f x; g y;` est généré, ce qui est incorrect
- car cette sous-séquence exécute 2 fois `g y`.
- solution : suspendre les écritures dans le journal quand une fonction journalisée est en cours d'exécution

Appels de fonctions elle-même journalisées

code

```
(* true ssi une fonction journalisée est en cours
   d'appel *)
let started = ref false

let register name ty_x f x =
  if !started then f x
  else begin
    Sentences.add (print_sentence name ty_x x);
    started := true;
    f x;
    started := false
  end
```

Et si `f x` lève une exception ?

Exceptions

Si l'appel d'une fonction journalisée lève une exception :

- si elle est rattrapée ensuite, le code généré doit aussi la rattraper
- si elle n'est pas rattrapée
  - comportement erronée de l'application (Frama-C)
  - prévenir proprement l'utilisateur
- exemple (`f` et `g` sont journalisées, pas `h`) :

```
let f () = ... raise E ...
let g () = ...
let h () = try ... f () ... with E → g ()
```

si `h` est exécutée et `E` levée, le journal doit contenir :

```
try f (); assert false with exn (* E *) → g ()
```

Exceptions

Code

```
let register name ty_x f x =
  try
    f x;
    Sentences.add (print_sentence name ty_x x);
  with exn →
    Sentences.add
      (fun fmt →
        Format.fprintf fmt
          "try %t assert false with exn (* %s *) → "
          (print_sentence name ty_x x)
          (Printexc.to_string exn));
    raise exn
```

générer le code après l'exécution de la fonction

Fonctions multi-arguments

Comment journaliser `f:` $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \tau_n \rightarrow$ `unit` ?

- retarder l'écriture d'un appel à `f` jusqu'à son application totale
- hypothèse : pas d'effet de bord au cours des applications partielles
- accumuler dans une continuation l'écriture des arguments

```
let extend_continuation f_acc pp_arg arg fmt =
  Format.fprintf fmt "%t %a" f_acc pp_arg arg
```

- itérer récursivement sur le type de `f`
  - type non fonctionnel : appliquer la continuation
  - type fonctionnel : étendre la continuation
  - utilisation d'un GADT pour distinguer les cas

Fonctions multi-arguments

Code

```
let rec journalize:
  τ. (Format.formatter → unit) → τ Type.t → τ → τ =
  fun (type t) f_acc (ty: t Type.t) (x:t) →
    match Type.Function.gadt_instance ty with
    | Type.Function.No_instance →
      Sentences.add (print_sentence f_acc ty x);
      x
    | Type.Function.Instance(ty_y, ty_res) →
      fun y →
        let res = x y in
        let f_acc =
          extend_continuation f_acc (pp ty_y) y in
        journalize f_acc ty_res res
```

Valeur de retour

Comment journaliser `f` : $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \tau_n$?

- la valeur `v` retournée peut être utilisée en argument d'une autre fonction `g` journalisée
- afficher `v` *via* `Datatype.pretty_code` est alors incorrect :
  - une nouvelle valeur est construite
  - incompatible avec l'égalité physique `==`
- solution : génération d'une liaison locale

```
let v = f x in
... g v; (* ou: let v' = g v in *) ...
```

- utilisation d'une table associant un nom de variable à une valeur OCaml

Valeur de retour

Code

```
let print_sentence f_acc ty x fmt =
  let pp fmt = Format.fprintf fmt "%s" f_acc in
  if Type.equal ty Datatype.unit then
    (* comme précédemment *)
    pp fmt
  else
    (* ajoute une liaison locale *)
    let v = gen_binding () in
    Binding.add ty x v;
    Format.fprintf fmt "let %s = %t in " v pp
```

frama C
Software Analyzers

Valeur de retour

Code (fonction d'affichage d'un argument)

```
let pp ty fmt x =
  try
    let name = Binding.find ty x in
    (* utiliser la chaîne pré-enregistrée
       si elle existe *)
    Format.fprintf fmt "%s" name
  with Not_found →
    (* utiliser l'afficheur par défaut sinon *)
    Datatype.pretty_code ty fmt x;
```

Conclusion

journal = script généré, reproduisant les actions utilisateurs

- très peu invasif
- surcoût temps/mémoire négligeable (pour Frama-C)
- fonctions monomorphes
  - pas d'effets de bord entre applications partielles
  - arguments fonctionnels
  - labels et arguments optionnels
- *pretty printing*
  - parenthésage minimal
  - indentation
  - commentaires

http://frama-c.com

Déclaration du GADT

```
type _ gadt_instance =
  | No_instance
  | Instance: α Type.t × β Type.t
      → (α → β) gadt_instance
```

- `Type.Function.gadt_instance: α Type.t → α gadt_instance` ne renvoie le constructeur `Instance` que si elle est appelée avec une valeur d'un type $(\tau_1 \to \tau_2)$ `Type.t`.
- dans ce cas, garantie statique que les première et deuxième composantes du constructeur sont respectivement des valeurs de type τ_1 `Type.t` et τ_2 `Type.t`.
- l'implémentation de `Type.Function.gadt_instance` garantie la réciproque