Chamelon : un minimiseur pour et en OCaml

Milla Valnet, Nathanaëlle Courant, Guillaume Bury, Pierre Chambart, Vincent Laviron

Chamelon : un minimiseur pour et en OCaml

Milla Valnet[1,2,3], Nathanaëlle Courant[3], Guillaume Bury[3], Pierre Chambart[3] et Vincent Laviron[3]

[1]École Normale Supérieure, Université PSL, Paris, 75005, France
[2]Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
[3]OCamlPro, Paris, 75014, France

Lors du développement d'outils manipulant du code OCaml, il arrive que ceux-ci échouent sur un programme. Identifier et comprendre l'erreur passe généralement par réduire à la main la taille du programme en question, de sorte à obtenir un programme plus court provoquant la même erreur — tâche souvent longue, parfois complexe, rarement intéressante. Nos travaux consistent à automatiser cette tâche à l'aide d'un minimiseur, ou *delta-débugueur*. Pour ce faire, nous proposons une liste d'heuristiques unitaires, ie. appliquant au programme des réductions de faible ampleur, et un algorithme itératif pour les combiner. Ces propositions sont implémentées dans l'outil libre Chamelon. Bien que pensé pour assister le développement d'un compilateur OCaml, ce dernier s'adapte à toutes sortes de projets manipulant du code OCaml. Il permet d'analyser des projets composés de plusieurs fichiers et minimise efficacement des programmes réels/concrets, réduisant leur taille d'un à plusieurs ordres de grandeur. Il est actuellement utilisé pour assister le développement du compilateur optimisant flambda2.

1 Introduction

Les erreurs de programmes se produisent parfois sur des codes longs, très longs, de plusieurs centaines voire milliers de lignes. Identifier et isoler l'erreur est une tâche souvent longue et fastidieuse, qui consiste généralement à minimiser manuellement la taille du programme autant que possible, en supprimant des lignes, simplifiant des expressions, etc. L'objectif d'un minimiseur est alors d'automatiser ce travail.

L'idée de la minimisation n'est pas nouvelle. Parfois appelée réduction, ou encore delta-debugging en ce qu'elle travaille à la jonction, dans le « delta », entre programmes avec et sans erreur, elle est développée dès 1999 par Andreas Zeller [11] avec pour but d'isoler la cause d'une erreur de programmes en y appliquant itérativement des simplifications de parties inutiles du programme. Elle y est définie comme la méthodologie pour « réduire un problème tout en préservant une certaine propriété » — ici, l'erreur. L'outil ainsi construit ne supprime pas l'erreur, mais au contraire la pointe.

Cette méthode est déjà utilisée pour des langages comme C, avec C-reduce [1], SMT-lib, avec ddSMT [6], ou encore via de nombreuses implémentations des travaux originaux de Zeller [11]. Néanmoins, cette problématique demeure très étudiée. Zeller travaille avec Hildebrandt [10] pour identifier les entrées et interactions provoquant l'échec du programme,

avec comme cas d'étude des entrées utilisateurs sur le navigateur de Mozilla, puis démontre avec Cleve [3] que le delta-debugging fonctionne aussi bien pour identifier les erreurs dues au code lui-même qu'à ses paramètres. Dans une volonté de voir les tâches de débugage comme des cas particuliers de problèmes de minimisation, il utilise cette méthode avec Choi [2] pour les échecs d'ordonnancement de fils, ou encore avec Cleve [4] pour identifier quelles variables causent l'erreur, et à quel pas d'exécution. Enfin, Leitner et al. [7] combinent cette approche avec du découpage de programmes (*program slicing*) pour réduire la taille des cas d'échecs dans le cadre de la génération aléatoire de tests.

Pour améliorer l'état de l'art en delta-debugging, certains tentent d'utiliser de l'apprentissage machine [5], d'autres encore des algorithmes probabilistes [9], etc. En OCaml cependant, les outils existants se limitent à tenter de comprendre la cause d'une erreur de type [8].

Cependant, aucun tel outil généraliste n'existe pour le langage OCaml. L'objectif de ce projet est ainsi de proposer le premier minimiseur pour du code OCaml, Chamelon. Bien qu'initialement pensé pour assister le développement d'un compilateur OCaml, un tel outil peut se révéler utile pour le développement d'autres projets utilisant ou manipulant du code OCaml. Son fonctionnement ne repose ni sur de l'apprentissage machine, ni sur une approche probabiliste, et est exclusivement déterministe. Ce travail fournit donc les contributions suivantes :

- une liste d'heuristiques visant à minimiser le programme OCaml en entrée ;
- un algorithme simple et efficace permettant de les itérer ;
- une implémentation en OCaml d'un minimiseur appliquant ces heuristiques, supportant les projets multifichiers et les erreurs à runtime, et adaptables à moindre coût à différentes versions du compilateur.

Plan. La section 2 présente les heuristiques unitaires proposées pour minimiser le programme, tandis que la section 3 explicite la manière dont celles-ci sont combinées. La section 4 montre comment ce travail s'étend à un projet composé de plusieurs fichiers, mais aussi comment il peut être utilisé lorsque l'erreur est déclenchée non plus à la compilation, mais à l'exécution du programme. La section 5 présente les résultats expérimentaux.

2 Les heuristiques

Le concept de l'approche est de composer et combiner différentes heuristiques unitaires, en appliquant chacune d'entre elles autant que possible. Nous présentons ici les différentes heuristiques proposées pour minimiser un programme OCaml. Notons que, ciblant initialement des problèmes de compilation, notre approche vise bien plus à identifier des erreurs causées par une certaine structure de code bien plus que par une certaine sémantique ou une certaine exécution : cela guide donc nos choix d'heuristiques.

2.1 Supprimer des définitions

Supprimer les définitions en commençant par la fin

Les minimisations les moins raffinées s'intéressent à la simple suppression d'éléments du code. La première heuristique consiste ainsi à supprimer toute définition — de variables, de types, de modules, etc. — en partant de la fin. De la sorte, on cherche à supprimer le code situé après la cause de l'erreur, dont l'erreur ne dépend pas.

Remplacer les expressions par des valeurs triviales

Lorsque certaines définitions ne peuvent être simplement supprimées, on choisit de poursuivre la minimisation du code en simplifiant au maximum toutes les expressions présentes dans le code. Pour ce faire, on tente de les remplacer par des valeurs les plus

simples possibles. L'enjeu est alors de déterminer par quelle valeur triviale on souhaite remplacer notre expression tout en respectant la contrainte de type sur cette dernière.

Pour les types de base, on remplace simplement les expressions de type `int` par 0, celles de type `float` par 0.0, celles de type `char` par `'0'`, celles de type `string` par `""` et celles de type `unit` par `()`. Pour les autres types, nous avons opté pour une solution générale et nous basons ainsi sur les deux définitions suivantes :

```
let __dummy1__ () = assert false [@@inline never]
external __dummy2__ : unit -> 'a = "%opaque"
```

En effet, `__dummy1__ ()` et `__dummy2__ ()` sont des expressions de type `'a`, qui peuvent ainsi remplacer une expression de n'importe quelle type.

Remarquons que nous aurions pu définir `let __dummy1__ = assert false`. Cependant, cette ligne de code entraînerait un échec à son exécution : pour cette raison, un compilateur risque alors d'optimiser le code produit en propageant cet échec d'assertion, le modifiant ainsi significativement, ce qui n'est pas souhaitable en delta-debugging. En enveloppant l'assertion dans une fonction, on retarde ainsi son exécution, prévenant cette optimisation. En ajoutant `[@@inline never]`, on empêche également que cette assertion soit étendue (*inlinée*) aux sites d'appels de `__dummy1__`, où elle entraînerait une nouvelle propagation de l'échec d'assertion. [1]

La définition de `__dummy2__` se repose quant à elle sur la primitive externe `opaque` : lors de la compilation, celle-ci est considérée comme une fonction renvoyant une valeur arbitraire — ici, comme une fonction de type `unit -> 'a` en raison de l'annotation. Cependant, à l'exécution, cette dernière se comporte comme la fonction identité. Pour cette raison, à l'exécution, la valeur de `__dummy2__ ()` sera `()`, causant une erreur de type.

Ainsi, dans les deux cas, l'utilisation de ces expressions *dummy*, triviales, entraîne une erreur à l'exécution du code produit. Lorsque l'objectif est d'assister la résolution d'échecs à la compilation, ce n'est pas une limitation. Cependant, à des fins de généralisation des cas d'utilisation de l'outil, cette problématique sera traitée dans la partie 4.

2.2 Simplifier des types

On cherche ensuite à simplifier les types construits définis par l'utilisateur.

Supprimer les constructeurs de types construits

Une première heuristique consiste à supprimer un constructeur `Cons` d'un type algébrique. Il s'agit alors de propager dans le code la suppression du constructeur : les expressions sous la forme `Cons`$(e_1, \ldots, e_n)$ sont remplacées par `__dummy1__ ()` ou `__dummy2__ ()`, et pour les motifs utilisant constructeur `Cons`, on supprime simplement le cas de filtrage concerné.

Supprimer les champs des types enregistrement ou des constructeurs

Lorsque supprimer le constructeur entier n'est pas possible, une heuristique peut à la place tenter d'en supprimer des champs. Ainsi, on supprime le i-ème champ de sa définition, et on parcourt à nouveau le code pour supprimer le i-ème champ dans chaque expression de la forme `Cons(e1,..,en)`, et le i-ème sous-motif de chaque motif de la forme `Cons(p1,..,pn)` — on veille alors à remplacer dans l'expression qui suit toutes les variables liées par `pi` par `__dummy1__ ()` ou `__dummy2__ ()`.

1. Notons qu'un compilateur pourrait utiliser l'information que `__dummy1__` ne renvoie jamais de valeur, ce qui influerait sur le code produit — ce n'est cependant pas le cas du compilateur sur lequel nous travaillions.

2.3 Simplifier le code

Modifier les attributs

On cherche en premier lieu à retirer tous les attributs de fonctions, modules, etc. du programme de sorte à le rendre moins verbeux. Cependant, ajouter certains attributs peut également fournir des informations précieuses sur l'origine de l'échec. En conséquence, l'une des heuristiques tente d'ajouter `local[never|always]` et `inline[never|always]` aux fonctions du programme, de sorte à fixer les stratégies d'*inlining* du compilateur.

'Etendre les fonctions

'Etendre une fonction, ie. la remplacer par sa définition à un site d'appel, peut permettre des simplifications supplémentaires au dit site. Notons que cette transformation diffère en deux points de l'ajout de l'attribut `[@@inline always]`, qui indique au compilateur d'effectuer l'extension (ou *inlining* : tout d'abord, en effectuant la transformation « à la main », cela permet à Chamelon d'effectuer des minimisations supplémentaires à cet emplacement. Enfin, si la cause de l'échec du programme provient, par exemple, de l'utilisation d'attribut, ces deux heuristiques auront des effets différents.

Aplatir les modules

Aplatir les modules correspond à sortir les définitions de variables du bloc `module Name = struct ... end`. Ce faisant, il faut néanmoins prendre garde à éviter les conflits de noms entre des variables définies dans le module et des variables définies dans le programme. À cette fin, nous avons choisi de faire précéder le nom de la variable par le nom du module dont elle est initialement issue : il faut ensuite propager ce changement dans le programme.

2.4 Retirer les artefacts de simplification

Après avoir appliqué les heuristiques ci-dessus, des artefacts peuvent apparaître, ie. des situations qui n'apparaîtraient pas ou peu dans un code utilisateur.

Supprimer le code mort

Pour chaque variable, module, type, on parcourt le code de sorte à déterminer s'ils sont utilisés dans le programme. S'ils ne le sont pas, on supprime simplement leur définition.

Simplifier les filtrages par motif

Lorsque le filtrage ne contient qu'un seul motif constitué d'une variable, on remplace `match e1 with x -> e2` par `e2` dans lequel `x` a été substituée textuellement par `e1`.

Simplifier les séquences

On remplace les expressions sous la forme `() ; e` par `e`.

Séquentialiser les appels de fonction

À la suite de simplifications, on peut obtenir une application de fonction sous la forme `(__dummy__ ()) e1 ... en`. On séquentialise alors en évaluant séparément chaque argument, de sorte à obtenir des expressions non imbriquées. On utilise la primitive suivante [2] :

```
external __ignore__ : 'a -> unit = "%ignore"
```

2. Cette primitive existe déjà avec ce type dans la bibliothèque standard OCaml. La redéfinir permet cependant de ne pas en dépendre.

On transforme alors `(__dummy__ ()) e1 ... en` en :
`__ignore__ e1 ; ... ; __ignore__ en ; __dummy__ ()`

Simplifier les `rec` et les arguments inutilisés

À la suite du remplacement d'expressions et de définitions par les valeurs `dummmy` détaillées plus haut, il arrive que des arguments d'une fonction ne soient plus utilisés dans son corps. Dans ce cas, on cherche à supprimer ces derniers.

Il faut alors prendre garde à propager leur suppression à tous les sites d'appel de la fonction en question. Lorsque le i-ème argument de la fonction `f` est supprimé, on remplace alors toutes les occurrences `f` par `(fun x1 ... xn -> f x1 ... xi-1 xi+1 ... xn)`.

De même, lorsque la fonction n'est plus récursive, on retire le mot clé `rec`.

3 L'itération

Une fois muni de ces heuristiques, il s'agit de les itérer astucieusement de sorte à minimiser efficacement un programme.

3.1 Interface avec les heuristiques

Une heuristique unitaire peut généralement s'appliquer en différents points d'un programme. Elle est implémentée de sorte que, étant donné un indice i, elle tente de minimiser le i-ème point qu'elle peut simplifier. Trois cas se présentent alors :

— Cette simplification ne supprime pas l'erreur : on a minimisé le programme !

— Cette simplification supprime l'erreur : on ne souhaite pas l'appliquer.

— L'indice est plus grand que celui du dernier point qu'on peut modifier.

On définit un type algébrique correspondant pour le type de retour d'un minimiseur :

```
type 'a minimized_step_result =
  | New_state of 'a
    (* New (smaller) states that produces an error *)
  | Change_removes_err
    (* This change removes the error, but other might be possible *)
  | No_more_changes
    (* The last possible position for changes has been reached *)
```

3.2 Itération linéaire

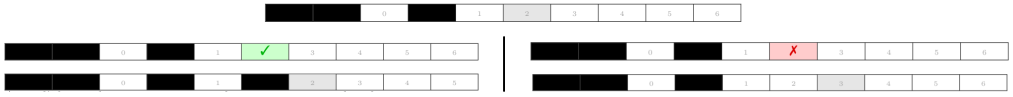
On dispose d'un état `state` qui représente notre programme et une heuristique `f`. Il s'agit désormais d'itérer l'application de cette heuristique sur le programme :

```
let minimize_basic (state : 'a)
    (f : 'a -> pos:int -> 'a minimized_step_result) : 'a * bool =
  let rec aux (state : 'a) (pos : int) (ever_changed : bool) =
    match f state ~pos with
    | New_state nstate -> aux nstate pos true
    | Change_removes_error -> aux state (pos + 1) ever_changed
    | No_more_changes -> (state, ever_changed)
  in
  aux state 0 false
```

Ici, la fonction `aux` prend en paramètre le programme, une position à laquelle tenter d'effectuer les changements, et un booléen symbolisant le fait que de tels changements aient eu lieu précédemment. Elle renvoie l'état du programme après application itérée de l'heuristique et ce booléen.

Le fonctionnement est le suivant : on tente de minimiser le programme au point donné. Si la minimisation est possible, on itère sur le nouvel état, en signifiant que des changements ont été effectués ; il n'y a nul besoin d'incrémenter la position, puisqu'après avoir simplifié le n-ième point, le point suivant est devenu n-ième sur le nouveau programme. Si la minimisation n'est pas possible, le programme et le booléen ne changent pas et on examine la position suivante. Enfin, si la position est trop grande, cela signifie que le programme a été parcouru, et on s'arrête donc en renvoyant le programme et le booléen courant.

On peut se représenter le fonctionnement de cet algorithme grâce à ce schéma :



Ainsi, l'idée est la suivante : on considère un programme dont les cases noires sont des points du programme déjà minimisés et les cases blanches sont des points non minimisés. On tente de minimiser la case 2, en gris, ie. le troisième point où peut s'effectuer une minimisation. Deux cas se présentent alors :

— La minimisation ne retire pas l'erreur (à gauche) : on l'applique, la case devient noire, et dans ce cas, on considère à nouveau le troisième point du programme où peut s'effectuer une minimisation.

— La minimisation retire l'erreur (à droite) : on ne souhaite pas l'appliquer, et dans ce cas, on considère la case 3, ie. le quatrième point du programme où peut s'effectuer une minimisation.

On remarque qu'un tel itérateur est modulaire en l'heuristique `f` passée en paramètre : il est donc aisé d'ajouter de nouvelles heuristiques.

3.3 Optimisation dichotomique

Une telle boucle s'optimise aisément par dichotomie, en ne tentant plus de minimiser des emplacements un par un, mais plutôt un ensemble d'emplacements d'une longueur 2^n. Cette méthode permet de gagner un facteur 10 d'efficacité sur des programmes réels de quelques milliers de lignes.

3.4 Ordre des heuristiques

L'ordre dans lequel combiner les différentes heuristiques pour optimiser le temps de minimisation a été déterminé expérimentalement, sur un petit échantillon de tests, en se contentant principalement de terminer par les heuristiques retirant les artefacts de simplification. Pour un ordonnancement plus robuste et efficace, une recherche plus poussée et des tests supplémentaires pourraient se révéler utiles.

4 Extensions

Jusqu'alors, nous nous sommes concentrés sur la minimisation d'un programme unique dont la compilation échoue. Cependant, des extensions ont été réalisées de sorte à en élargir le cadre d'utilisation.

4.1 Multifichier

Dans des cas d'utilisation réels, le travail s'effectue sur un projet composé d'un ensemble de fichiers, dépendant les uns des autres. Ainsi, nous avons cherché à adapter Chamelon de sorte à le rendre capable d'opérer sur des projets composés de plusieurs fichiers. On ajoute ainsi quelques minimisations spécifiques au cas multifichier :

- Tout d'abord, on cherche à supprimer le maximum de ces fichiers, dans l'ordre des dépendances ;
- ensuite, on tente de fusionner des fichiers pour aboutir au nombre minimal de fichiers à minimiser ;
- une fois ces deux minimisations effectuées, on minimise indépendamment chaque fichier restant avec les méthodes montrées précédemment.

Lors de ce dernier point, une précaution toute particulière doit être prise : en effet, lorsqu'un objet est modifié par une heuristique, il faut prendre bien garde à en propager les modifications à toutes ses dépendances : par exemple, si un argument de la fonction `f` est supprimé, il faut supprimer cet argument à tous les points d'appels de `f`, dans chacune des dépendances du programme.

Pour utiliser Chamelon en multifichier, il faut lui fournir la liste des fichiers sur lesquels effectuer la minimisation, dans l'ordre des dépendances — qui peut être obtenu à l'aide de l'outil `ocamldep`.

4.2 Runtime

Les travaux présentés jusqu'à présent se concentrent sur des erreurs à la compilation. Cependant, il arrive que l'erreur soit provoquée non pas à la compilation, mais à l'exécution du programme, via notamment une erreur de segmentation. En conséquence, on adapte les heuristiques proposées.

En effet, l'utilisation des variables `__dummy1__` et `__dummy2__` provoque des erreurs à l'exécution, rendant le minimiseur inutilisable lorsque le programme doit être exécuté.

L'enjeu est donc d'écrire un algorithme qui, étant donné un type d'entrée, génère une expression de même type, la plus simple possible. Il se construit de la manière suivante :

```
let rec generate_dummy_expr env typ =
  match typ with
    | Tint -> Texp_constant (Const_int 0)
    | Tvar vtyp -> (* 'a *)
      let id = find_value_of_typ vtyp env in Texp_ident id
    | Tarrow (arg_label, t1, t2) -> (* t1 -> t2 *)
      let param = Ident.create "x" in
      let nenv = add_value param t1 env in
      Texp_function({arg = param ; body = generate_dummy_expr env t2})
    | Tconstr (tname, t_list, _) -> (* t1 * ... * tn tname *) ->
      let type_descr = find_type tname env in
      let non_rec_cons = List.filter non_recursive type_descr.constrs in
      let cons =  arity_min non_rec_cons in
      let arg_types = instantiate cons.args t_list in
      Texp_construct(cons, List.map (generate_dummy_expr env) arg_types)
```

Notons que nous présentons ici une version simplifiée de l'AST à des fins pédagogiques. Son fonctionnement est le suivant : lorsque le type est `int`, `char`, `string` ou `unit`, on génère une expression simple du type souhaité, ie. `0`, `'0'`, `""` ou `()`. Lorsque le type est une variable de type (par exemple, à l'intérieur d'une fonction polymorphe), on cherche

dans l'environnement une variable de ce type et on la renvoie. Lorsque ce type est un type flèche `t1 -> t2`, on ajoute dans l'environnement une variable de type `t1`, on construit une expression `e` de type `t2` dans cet environnement, et on renvoie `fun x -> e`.

Enfin, lorsque ce type est un type construit `(t1, ..., tn) tname`, on cherche la définition de ce type dans l'environnement. On construit ensuite une expression avec le constructeur `Cons` d'arité minimale du type `('a1, ..., 'an) tname` ne contenant pas d'appel récursif au type `tname`. Pour ce faire, on considère le type des arguments de ce constructeur et on prend garde d'en instancier les variables de type `a1`, ..., `an` par `t1`, ... `tn`. Enfin, on génère une expression `ei` du type de chaque argument et on renvoie `Cons(e1, ... , em)`.

Remarquons qu'un tel algorithme peut échouer (s'il n'existe pas de constructeur non récursif ou une variable d'un certain type) : dans ce cas, l'expression n'est pas simplifiée. Dans le cas contraire, on simplifie sans recours à `__dummy1__` ou `__dummy2__` : ainsi, minimiser un programme n'introduit pas d'erreur supplémentaire à l'exécution. Selon le contexte, l'utilisateur peut ainsi choisir entre les heuristiques initiales, plus efficaces mais introduisant de telles erreurs, et cette heuristique.

4.3 Bibliothèque de compatiblité

L'implémentation se base sur l'utilisation des compiler-libs d'OCaml, afin de lire le `.cmt` produit à la compilation d'un programme, et de manipulerl'arbre de syntaxe abstrait typé qu'elle en obtient (via `Tast_mapper`). Cependant, cette bibliothèque est différente selon le compilateur ou sa version.

En conséquence, une bibliothèque de compatibilité a été implémentée : ainsi, changer de version de compilateur s'effectue à moindre coût, requérant simplement quelques informations sur la nouvelle forme de l'AST.

5 Implémentation et évaluation expérimentale

Les différentes méthodes et heuristiques explicitées plus haut ont toutes été implémentées dans l'outil Chamelon, disponible en source libre sur github [3].

5.1 Contexte de développement

À l'origine, Chamelon avait pour objectif d'assister le développement du compilateur optimisant flambda2 [4]. En effet, lorsque celui-ci échouait sur un programme ou ensemble de programmes pourtant corrects au regard du compilateur standard, identifier la cause de l'erreur dans flambda2 n'était pas toujours aisé.

Cependant, Chamelon est construit de manière modulaire. S'il applique les minimiseurs décrits plus hauts en s'assurant qu'une certaine condition demeure valide, il est parfaitement modulaire en la dite condition. Il suffit alors de fournir à l'outil la commande que l'on souhaite exécuter sur le programme, et la condition/l'erreur que l'on souhaite maintenir en résultat. Il faut néanmoins noter que les transformations effectuées par Chamelon s'intéressent plus à la structure du programme et ne maintiennent pas sa sémantique. En conséquence, l'utiliser sur des outils dépendant fortement de données sémantiques comme la valeur des variables produirait vraisemblablement des résultats limités.

5.2 Résultats expérimentaux

L'outil ainsi développé est actuellement utilisé par l'équipe flambda à OCamlPro pour assister la production de flambda2. Il y a obtenu des résultats probants sur des cas réels d'échec de celui-ci, diminuant significativement la taille du problème en sortie.

3. disponible au lien https://github.com/ocaml-flambda/flambda-backend/tree/main/chamelon
4. disponible au lien https://github.com/ocaml-flambda/flambda-backend

Parmi les résultats expérimentaux, le minimiseur a été en mesure de minimiser un programme de 650 lignes dont la compilation échouait en un programme minimisé de seulement 6 lignes provoquant la même erreur, permettant d'identifier un problème lié à l'optimisation des filtrages par motifs :

```
let offset ~byte_order  byte_nr =
  match byte_order with | `Little_endian -> 0 | `Big_endian -> byte_nr
let pack_unsigned_16 ~byte_order  =
  __ignore__ ((offset ) ~byte_order 0);
  __ignore__ ((__dummy2__ ()) ((offset ) ~byte_order 1));
  __dummy2__ ()
```

Nous avons également testé le minimiseur sur des programmes de taille plus conséquente. Par exemple, étant donné un des fichiers du typeur Caml de 3842 lignes, sur lequel le compilateur échouait, le minimiseur a été en mesure de le réduire à un programme de 22 lignes seulement, le tout en une trentaine de minutes sur un ordinateur portable moyen :

```
open Types
module String = Misc.Stdlib.String
type module_entry = | Mod_persistent
and module_data = {  mda_declaration: Subst.Lazy.module_declaration }
and t = { local_constraints: type_declaration Path.Map.t }
let persistent_env : module_data Persistent_env.t ref = __dummy2__ ()
  [@@local never][@@inline never]
let find_pers_mod name =
  Persistent_env.find (!persistent_env) (__dummy2__ ()) (__dummy2__ ())
  [@@local never][@@inline never]
type _ load =
  | Don't_load: unit load
  | Load: module_data load
let lookup_ident_module (type a) (load : a load) =
  match __dummy2__ () with
  | Mod_persistent ->
      (match load with
       | Don't_load -> (__dummy2__ (), __dummy2__ ())
       | Load ->
           (match find_pers_mod (__dummy2__ ()) with
            | mda -> ((__dummy2__ ()), (mda : a))
            | exception Not_found -> __dummy2__ ()))[@@local never][@@inline never]
```

Remarquons que la réduction de la taille du programme n'est pas la seule action intéressante du minimiseur. En effet, sur cet exemple, on peut noter qu'à la ligne 18, (`__dummy2__`(), `__dummy2__`()) n'a pas été simplifié en `__dummmy2__`() : cela signifie que cette modification retirait l'erreur, information qui peut donc être exploitée pour la comprendre. Il s'agit de ne pas seulement tirer avantage de la minimisation, mais aussi de la *minimalité* du programme au regard des heuristiques appliquées.

Souvent, la sortie peut encore être minimisée à la main. Cependant, le minimiseur réalise une partie conséquente du travail de manière automatique. Enfin, dans le cas d'erreurs dans un cadre multifichier, le minimiseur est bien en mesure de fusionner ou supprimer des fichiers, de sorte à aboutir à une copie minimisée du projet qui déclenchait l'erreur.

Conclusion

À l'avenir, une extension intéressante serait de rendre le minimiseur Chamelon compatible avec un projet `dune`, et non plus seulement avec une liste de fichiers. Enfin, notamment à travers son utilisation dans des exemples réels, il serait intéressant d'améliorer les heuristiques existantes ou d'en trouver de nouvelles, de sorte à rendre le minimiseur plus robuste, plus efficace et plus rapide.

Néanmoins, ce travail propose le premier minimiseur, ou delta-débugueur pour et en OCaml, au service de sa communauté !

Références

[1] *C-reduce project*. URL : https://github.com/csmith-project/creduce.

[2] Jong-Deok Choi et Andreas Zeller. "Isolating failure-inducing thread schedules". In : *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 2002, p. 210-220.

[3] Holger Cleve et Andreas Zeller. "Finding Failure Causes through Automated Testing". In : *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*. Sous la dir. de Mireille Ducassé. 2000. URL : https://arxiv.org/abs/cs/0012009.

[4] Holger Cleve et Andreas Zeller. "Locating causes of program failures". In : *Proceedings of the 27th international conference on Software engineering*. 2005, p. 342-351.

[5] Kihong Heo et al. "Effective Program Debloating via Reinforcement Learning". In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada : Association for Computing Machinery, 2018, p. 380-394. ISBN : 9781450356930. DOI : 10.1145/3243734.3243838. URL : https://doi.org/10.1145/3243734.3243838.

[6] Gereon Kremer, Aina Niemetz et Mathias Preiner. "ddSMT 2.0 : Better Delta Debugging for the SMT-LIBv2 Language and Friends". In : *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Sous la dir. d'Alexandra Silva et K. Rustan M. Leino. T. 12760. Lecture Notes in Computer Science. Springer, 2021, p. 231-242. DOI : 10.1007/978-3-030-81688-9_11. URL : https://doi.org/10.1007/978-3-030-81688-9%5C_11.

[7] Andreas Leitner et al. "Efficient Unit Test Case Minimization". In : *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA : Association for Computing Machinery, 2007, p. 417-420. ISBN : 9781595938824. DOI : 10.1145/1321631.1321698. URL : https://doi.org/10.1145/1321631.1321698.

[8] Joanna Sharrad et Olaf Chitil. "Refining the Delta Debugging of Type Errors". In : *Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages*. IFL '21. Nijmegen, Netherlands : Association for Computing Machinery, 2022, p. 10-19. ISBN : 9781450386449. DOI : 10.1145/3544885.3544888. URL : https://doi.org/10.1145/3544885.3544888.

[9] Guancheng Wang et al. "Probabilistic Delta Debugging". In : *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece : Association for Computing Machinery, 2021, p. 881-892. ISBN : 9781450385626. DOI : 10.1145/3468264.3468625. URL : https://doi.org/10.1145/3468264.3468625.

[10] A. Zeller et R. Hildebrandt. "Simplifying and isolating failure-inducing input". In : *IEEE Transactions on Software Engineering* 28.2 (2002), p. 183-200. DOI : 10.1109/32.988498.

[11] Andreas Zeller. "Yesterday, My Program Worked. Today, It Does Not. Why ?" In : *SIGSOFT Softw. Eng. Notes* 24.6 (oct. 1999), p. 253-267. ISSN : 0163-5948. DOI : 10.1145/318774.318946. URL : https://doi.org/10.1145/318774.318946.