Patches Sémantiques pour OCaml

Fabrice Le Fessant[1] & Vincent Laviron[2]

1: Équipe Gallium, INRIA Paris,
75012, Paris, France
fabrice.le_fessant@inria.fr
2: OCamlPro SAS,
91190, Gif-sur-Yvette, France
vincent.laviron@ocamlpro.com

Résumé

Dans cet article, nous présentons l'utilisation qui est faite des patches sémantiques appliqués à OCaml chez OCamlPro, pour permettre de maintenir à moindre coût plusieurs branches d'OCaml intégrant des modifications importantes, en particulier pour l'ajout des informations dynamiques de types utilisées par l'OCaml Memory Profiler. Nous présentons les patches sémantiques que nous utilisons, leur implantation et nos retours d'expérience, ainsi que les perspectives d'utilisation de ces patches pour d'autres usages.

1. Introduction

Dans l'univers du logiciel libre, la maintenance de versions séparées d'un logiciel a toujours été considérée comme une tâche difficile et ingrate : si chaque version évolue de son coté, les resynchronisations nécessaires de temps en temps sont souvent laborieuses, car elles imposent d'appliquer des changements parfois volumineux en terme de lignes de code à modifier. Cela peut se traduire purement et simplement par l'abandon de ces tâches (absence de maintenance sur d'anciennes versions pour éviter d'y propager des patches), ou plus douloureusement par le retardement de ces synchronisations, jusqu'à ce qu'elles deviennent inévitables.

OCamlPro maintient ainsi plusieurs versions d'OCaml (4.01.0 et 4.02.1) sur lesquelles sont appliqués des modifications assez conséquentes pour permettre l'étude du comportement mémoire des applications. Ces modifications ne sont pas très complexes, mais très invasives, avec la propagation de nombreuses informations sur plusieurs passes de compilation. Pour diminuer le coût de maintenance et d'évolution de ces versions, une partie des modifications ont été implantées sous forme de *patches sémantiques*[3, 2]. Contrairement aux patches textuels classiques, qui sont très sensibles aux modifications concurrentes des sources, les patches sémantiques permettent de décrire des transformations des sources beaucoup plus résistantes, car ils s'appliquent non pas sur le texte source du programme, mais sur l'arbre de syntaxe déjà extrait.

A notre connaissance, il s'agit de la première utilisation des patches sémantiques pour un tel usage. Auparavant, les patches sémantiques étaient utilisés par des outils tels que Coccinnelle[1] pour effectuer des vérifications de correction sur un programme source. Ici, nous proposons de les utiliser directement pour programmer, et pour effectuer proprement du refactoring de code.

Dans cet article, nous commençons par présenter les principes régissant l'utilisation de nos patches sémantiques, puis nous décrivons leur implantation dans le contexte d'OCamlPro. Enfin, nous concluons en présentant de nouvelles perspectives d'utilisation de nos patches sémantiques, issues de notre utilisation courante.

2. Principes

Nos patches sémantiques définissent des transformations génériques de l'arbre de syntaxe d'OCaml. Un même patch sémantique peut engendrer de nombreuses modifications dans l'arbre de syntaxe, et il est possible, sous certaines conditions, de limiter ces modifications à une partie uniquement de l'arbre.

Les patches sémantiques que nous utilisons sont avant tout issus de nos besoins : ils correspondent à des patches textuels que nous avons développés, puis transformés en patches sémantiques pour en simplifier l'application sur de nouvelles versions d'OCaml. Par conséquent, nous montrons ici des exemples, qui ne sauraient former une liste exhaustive, même si la définition d'une telle liste revêt un grand intérêt pour la généricité de nos outils.

Nous allons commencer par un exemple de modification simple :

```
let type_exp env exp =
  match exp.pexp_desc with
  | ...
  | Pexp_seq(e1,e2) ->
    let e1 = type_exp env e1 in
    ...
```

devenant :

```
let type_exp loc env exp =
  match exp.pexp_desc with
  | ...
  | Pexp_seq(e1,e2,loc) ->
    let e1 = type_exp loc env e1 in
    ...
```

Ici, le programmeur a effectué plusieurs modifications : la fonction `type_exp` prend un premier argument `loc` qu'elle va propager aux autres fonctions qu'elle utilise, sauf quand cet argument est redéfini dans un cas particulier. C'est le cas ici du constructeur `Pexp_seq` qui possède maintenant un troisième argument, qui est lié dans le filtrage à une nouvelle variable `loc`. Ce code est certes artificiel, mais correspond à un très grand nombre de nos modifications. Il est important de remarquer que ces modifications sont globales, dans le sens où toutes les occurrences de `type_exp` et toutes les occurrences de `Pexp_seq` doivent être modifiées, aussi bien dans ce module que dans ceux qui en dépendent.

Nous pouvons maintenant introduire deux patches sémantiques :

— `add_function_argument` est le patch sémantique qui permet, pour une liste de fonctions `fun_names` de leur ajouter des arguments `fun_argnames` à une position `fun_argpos`. Les arguments sont des noms de variables, de façon à pouvoir apparaître aussi bien en tant que filtrages (dans les définitions) que dans des expressions (dans les applications) ;

— `add_constructor_argument` est le patch sémantique équivalent, pour une liste de constructeurs. Comme le précédent, ce patch ne s'applique pas sur la définition de type elle-même, sauf si on fournit aussi les types des arguments.

Ces deux patches sémantiques peuvent être appliqués sur l'exemple, sous la forme suivante :

```
patch add_function_argument {
  fun_names = ["type_exp"]
  fun_argnames = ["loc"]
  fun_argpos = 0
}
patch add_constructor_argument {
  constr_names = ["Pexp_seq"]
  constr_argnames = ["loc"]
  constr_argpos = 2
}
```

Une fois ces patches définis, ils seront automatiquement appliqués sur le code initial (version à gauche) par le compilateur pour générer le code binaire, correspondant à celui de droite.

Un second exemple est celui de l'extension d'un enregistrement avec un nouveau champ :

```
let prim_makearray =
  { prim_name = "caml_make_vect";
    prim_alloc = true;
    prim_native_name = ""; }
let prim_obj_tag =
  { prim_name = "caml_obj_tag";
   prim_alloc = false;
   prim_native_name = ""; }
```

devenant :

```
let prim_makearray =
  { prim_name = "caml_make_vect";
    prim_alloc = true; prim_memprof = true;
    prim_native_name = ""; }
let prim_obj_tag =
  { prim_name = "caml_obj_tag";
   prim_alloc = false; prim_memprof = true;
   prim_native_name = ""; }
```

Pour l'automatiser, nous définissons un nouveau patch sémantique :

```
patch add_record_field {
  record_labels = [ "prim_alloc" ]
  record_new_labels = [ "prim_memprof" ]
}
```

Ce patch contient une garde `record_labels` qui lui indique sur quels enregistrements il s'applique, et les labels qui doivent y être ajoutés. Ainsi, ici, la présence d'un label `prim_alloc` indique au compilateur qu'il va devoir ajouter aussi le label `prim_memprof`, si celui-ci n'est pas déjà défini. Nous utilisons la convention habituelle en OCaml que le champ prend alors la valeur de la variable de même nom dans l'environnement, i.e. `prim_memprof`. Celle-ci doit donc être définie dans l'environnement, ici avec la valeur `true`.

Prenons maintenant l'exemple de modification suivant :

```
type ulambda =
  Uvar of Ident.t
| Uconst of structured_constant * ...
| Udirect_apply of
   function_label * ulambda list * ...
| Ugeneric_apply of
   ulambda * ulambda list * ...
| ...
```

devenant :

```
type ulambda = { desc : ulambda_desc;
                 debug : Debuginfo.t; }
and ulambda_desc =
  Uvar of Ident.t
| Uconst of structured_constant * ...
| Udirect_apply of
   function_label * ulambda list * ...
| Ugeneric_apply of
   ulambda * ulambda list * ...
| ...
```

Dans cet exemple, nous avons emballé les constructeurs du type `ulambda` dans un enregistrement, de façon à ajouter à chaque noeud de l'arbre un champ `debug`. Cette transformation de la définition

de type est certes courte, mais elle se traduit par un patch textuel colossal, car tous les constructeurs de `ulambda` dans le reste du code doivent apparaître dans un tel enregistrement, qu'ils soient dans un filtrage ou dans une expression.

Pour éviter un tel patch, nous définissons alors le patch sémantique suivant :

```
patch box_constructors {
  box_constr_names = [ "Uvar"; "Uconst"; ... ]
  box_with_labels = [ "desc"; "debug" ]
}
```

Ce patch indique au compilateur que les constructeurs `Uvar`, `Uconst`, etc. doivent être emballés dans un enregistrement. Le premier label fournit indique le champ qui va contenir le constructeur emballé, ici `desc`, tandis que les autres champs fournissent des champs supplémentaires qui sont initialisés, comme dans le cas de `add_record_field`, par une variable de même nom dans l'environnement. Le patch sémantique ne modifie pas la définition de type, ce qui doit être fait séparément.

Une autre modification assez fréquente est de changer le nom d'un identificateur. On définit alors le patch sémantique suivant :

```
patch replace_idents {
  replace_idents = [ "former_name" ]
  replace_with = "new_name"
}
```

Enfin, une technique fréquente pour modifier le comportement d'une fonction est de la redéfinir dans un nouveau module, et de modifier tous les sites d'appel pour qu'ils utilisent la nouvelle fonction en place de l'ancienne. Cela peut se faire en ouvrant le nouveau module, qui cache ainsi l'ancienne définition :

```
patch open_modules {
  open_modules = [ "SafeString" ]
}
```

Ce patch est équivalent à l'ajout de `open SafeString` en tête de fichier. Le module ainsi ouvert peut redéfinir des modules ou des fonctions utilisés localement. En particulier, nous utilisons un module `SafeString` to provide full compatibility between 4.01.0 and 4.02.1 for `bytes` and `string`s. Any 4.02.1 module can become compatible with 4.01.0 by just using this semantic patch.

3. Mise en œuvre

La technologie des patches sémantiques a été introduite chez OCamlPro pour permettre la migration des patches liés au OCaml Memory Profiler depuis la version 4.01.0, utilisée dans la version publique en ligne, vers la version 4.02.1, proposée à nos clients dans le cadre de l'offre commerciale.

Une grande partie du nouveau code est placée dans des nouveaux modules externes, qui sont inclus dans les deux versions, grâce à l'utilisation d'un préprocesseur et de directives `#if OCAMLVERSION = "4.01.0+ocp1"... #else ... #endif`, intégrés dans le lexer du compilateur.

Le reste du code doit être inséré dans le code du compilateur de chaque version, pour soit appeler ces nouveaux modules, soit pour modifier le comportement des passes de compilation. L'intérêt des patches sémantiques est de réduire considérablement ces modifications.

Les patches sémantiques sont implantés comme des transformations sur l'arbre syntaxique, effectuées directement par le compilateur après le parsing. Ces transformations sont intégrées au compilateur, par compatibilité avec la version 4.01.0, mais il est prévu de les déplacer dans un `ppx` pour les versions ultérieures, dès que le support de la version 4.01.0 ne sera plus nécessaire.

Pour chaque fichier source compilé `myModule.ml`, le compilateur recherche les patches sémantiques associés dans un fichier `myModule.ml.ocpp`.

3.1. Intégration du support dans une nouvelle version

Une étape importante est la migration des patches sémantiques d'une version d'OCaml à l'autre. Celle-ci s'effectue donc en plusieurs étapes :

1. Intégration des patches et des nouveaux modules pour le support du préprocesseur dans la nouvelle version, puis bootstrap pour obtenir un compilateur intégrant ce support. Ces patches et les modules externes correspondant ne doivent ni dépendre du préprocesseur, ni des patches sémantiques.
2. Intégration des patches et des nouveaux modules pour le support des patches sémantiques dans la nouvelle version, puis bootstrap. Ce code effectuant des transformations sur l'arbre de syntaxe abstrait d'OCaml, qu'il aurait été pénible de maintenir pour chaque version d'OCaml, le choix a été fait d'utiliser les directives du préprocesseur pour factoriser au maximum le code commun, et dupliquer uniquement les différences de l'arbre de syntaxe abstrait entre les versions.

Un extrait d'un tel code est fourni par l'exemple suivant :

```
  let leave_pattern pat =
    match pat.ppat_desc with
#if OCAML_VERSION = "4.01.0+ocp1"
    | Ppat_construct({ txt = Lident constr} as loc, exp, bool) ->
#else
    | Ppat_construct({ txt = Lident constr} as loc, exp) ->
#endif
      begin try
        let (idents, used) = StringMap.find constr !box_constructors in
        ...
```

3. L'étape suivante est d'intégrer l'ensemble des patches restant dans la nouvelle version. Les fichiers correspondant aux patches sémantiques sont directement copiés dans la nouvelle version. Des patches textuels sont ensuite appliqués fichier source par fichier source, dans l'ordre d'intégration dans le compilateur.

Un ajout important a été la capacité du moteur d'application des patches sémantiques à signaler l'absence d'utilisation d'un patch sémantique : par exemple, un patch sémantique rajoutant un argument à une certaine fonction signale si la fonction n'a pas été trouvée. Cela permet d'éviter que, suite à un renommage de la fonction, le patch ne soit pas appliqué, silencieusement, par le compilateur.

4. Retours d'expérience

OCamlPro a initialement utilisé les patches sémantiques dans le contexte des versions Memprof des compilateurs OCaml. Nous avons maintenant étendu cet usage pour rendre compatible de façon transparente l'outil d'analyse des exceptions non rattrapées que nous développons.

4.1. Branches Memprof des compilateurs OCaml

Les deux versions d'OCaml actuellement maintenues avec le support pour le profilage mémoire contiennent chacune 52 patches sémantiques, dont la plupart s'appliquent à plusieurs endroits.

Ainsi, le fichier `pervasives.mli.ocpp` contient le patch suivant :

```
patch add_primitive_flags {
```

```
  prim_flags = [ "!memprof" ]
  prim_names = [ "caml_power_float" "caml_exp_float"
    "caml_expm1_float" "caml_acos_float" "caml_asin_float"
    "caml_atan_float" "caml_atan2_float" "caml_hypot_float"
    "caml_cos_float" "caml_cosh_float" "caml_log_float"
    "caml_log10_float" "caml_log1p_float" "caml_sin_float"
    "caml_sinh_float" "caml_sqrt_float" "caml_tan_float"
    "caml_tanh_float" "caml_ceil_float" "caml_floor_float"
    "caml_copysign_float" "caml_fmod_float" "caml_ldexp_float"
    "caml_float_of_string" ]
}
```

Celui-ci permet d'ajouter une annotation `!memprof` sur un certain nombre de primitives. Cette annotation indique au compilateur que la valeur allouée par la primitive a le même type que la valeur de retour de la primitive. Sans ce patch, chacune de ces primitives auraient dû être modifiée. De plus, le même patch est présent dans le fichier `pervasives.ml.ocpp`, ce qui permet de partager ces modifications entre les deux fichiers, chose beaucoup plus complexe à faire avec des patches textuels.

Les patches sémantiques sont très efficaces, mais souffrent à l'usage d'un inconvénient important : le code lu par le développeur est celui qui ne contient pas les patches, ce qui peut être assez perturbant, s'il ne connaît pas les patches sémantiques qui sont appliqués dessus. Ainsi, un argument supplémentaire à une fonction, qui n'apparaît que sous forme de patche sémantique, peut se traduire par l'utilisation de cet argument dans le corps de la fonction : le développeur peut alors ne pas comprendre la provenance de cette variable dans le corps de la fonction, puisqu'il ne voit pas qu'il s'agit d'un argument caché. Une solution à ce problème pourrait être de fournir un outil qui applique les patches sémantiques textuellement, de sorte que le développeur puisse alterner entre le code du fichier, qu'il peut modifier, et le code après application des patches sémantiques, qu'il ne peut modifier, mais qu'il peut utiliser pour comprendre le fonctionnement réel.

4.2. Outil d'analyse des exceptions

Nous avons récemment étendu l'application des patches sémantiques à l'outil d'analyse des exceptions non rattrapées, en cours de développement dans le projet SecurOCaml. Cet outil a été développé pour exploiter le *lambda code* généré par la version 4.01.0 du compilateur. Il s'agit du langage intermédiaire suivant immédiatement la passe de vérification des types du compilateur, et dont la structure est assez similaire à un lambda-calcul sans types.

Nous avons récemment souhaité tester l'outil sur des programmes compilés avec la version Memprof de notre compilateur 4.01.0, dont le lambda-code est légèrement différent, car il contient des identifiants permettant de retrouver les informations de types qui seront stockés dans les blocs alloués par le programme.

Pour permettre à l'outil d'être compatible avec les deux versions, 4.01.0 et 4.01.0+ocp1, notre première idée était d'ajouter des directives conditionnelles `#if OCAML_VERSION = "4.01.0"` dans le code, mais cela nous aurait imposé de distribuer notre préprocesseur indépendemment de notre compilateur, pour que nos partenaires du projet SecurOCaml puissent l'utiliser pour compiler les sources de l'analyseur.

A la place, nous avons choisi d'ajouter un fichier `.ocpp` qui contient les patches sémantiques permettant de modifier le code travaillant sur le lambda-code pour le rendre compatible avec notre propre version.

Cette solution présente l'avantage d'être totalement transparente pour nos partenaires : puisqu'ils n'utilisent pas notre compilateur, les patches sémantiques ne sont pas détectés, et leur compilateur peut correctement compiler le code pour la version standard 4.01.0. Au contraire, pous les développeurs d'OCamlPro, les patches sémantiques sont appliqués à la volée par le compilateur 4.01.0+ocp1, et l'application compile à nouveau sans aucune modification.

5. Perspectives

Ces premières expérimentations avec les patches sémantiques pour OCaml nous ont amenés à imaginer des extensions de ces patches sémantiques, ainsi que de potentiels nouveaux usages.

5.1. Amélioration du langage de patches

Le langage qui est utilisé par OCamlPro pour décrire les patches sémantique devrait être amélioré dans l'avenir pour permettre différentes améliorations :

Partage, nommage et extension des patches : Pour l'instant, chaque patch est placé dans le fichier `.ocpp` du fichier sur lequel il s'applique. Cela se traduit par une duplication inutile, et potentiellement dangereuse, des patches. Nous proposons donc d'étendre à l'avenir le langage pour :

Partager les patches : un fichier pourrait inclure un certain nombre de patches provenant d'un autre fichier ;

Nommer les patches : pour sélection les patches à inclurer, il devrait être possible de leur associer un identifiant ;

Étendre les patches : une fois un patch sélectionné depuis un autre fichier, il serait utile de pouvoir l'étendre pour s'appliquer au fichier courant.

Par exemple, si le fichier `patches.ocpp` contient la description suivante :

```
patch add_primitive_flags "add_memprof_flag" {
  prim_flags = [ "!memprof" ]
  prim_names = []
}
```

Le fichier `pervasives.mli.ocpp` pourrait ne contenir que :

```
import "../patches.ocpp"
patch "pervasives_add_memprof_flag" {
  extend "add_memprof_flag"
  prim_names +=  [ "caml_power_float" "caml_exp_float"
    "caml_expm1_float" "caml_acos_float" "caml_asin_float"
    "caml_atan_float" "caml_atan2_float" "caml_hypot_float"
    "caml_cos_float" "caml_cosh_float" "caml_log_float"
    "caml_log10_float" "caml_log1p_float" "caml_sin_float"
    "caml_sinh_float" "caml_sqrt_float" "caml_tan_float"
    "caml_tanh_float" "caml_ceil_float" "caml_floor_float"
    "caml_copysign_float" "caml_fmod_float" "caml_ldexp_float"
    "caml_float_of_string" ]
}
apply "pervasives_add_memprof_flag"
```

Et fichier `pervasives.ml.ocpp` pourrait ne contenir que :

```
import "pervasives.mli.ocpp"
apply "pervasives_add_memprof_flag"
```

On peut noter l'ajout d'un opérateur `apply` pour différencier la simple définition du patch, qui ne l'applique plus, de l'application. Le langage utilisé ici est purement indicatif, il ne sert ici qu'à montrer les fonctionalités que devrait posséder un tel langage.

Conditionnelles : le langage de patches devraient aussi probablement contenir des tests permettant de choisir les patches à appliquer en fonction de paramêtres extérieurs (version d'OCaml, etc.)

Contextualisation : enfin, les patches devraient pouvoir s'appliquer en fonction du contexte. Ainsi, aujourd'hui déjà, l'ajout d'un champ dans un enregistrement ne s'applique que si la présence d'un autre champ est détectée, et si le nouveau champ n'est pas déjà présent. Ce type de conditions sur le contexte devrait être possible à exprimer dans le langage de description des patches.

6. Format d'échange de modifications de sources

OCamlPro s'intéresse beaucoup aux outils d'aide au développement en OCaml. En particulier, beaucoup de jeunes développeurs sont habitués à l'environnement de programmation Eclipse de Java, qui fournit, pour chaque erreur ou avertissement, un ensemble de suggestions pour corriger le code. Il nous semble qu'une utilisation intéressante des patches sémantiques serait de pouvoir décrire complètement ces suggestions. Elles seraient ainsi transmises par l'outil de suggestion à l'éditeur, qui pourrait ensuite les soumettre à l'utilisateur pour choisir celle qu'il souhaite appliquer. Un autre outil serait alors en charge d'appliquer le patch sémantique choisi par l'utilisateur de façon textuelle sur le fichier. Cet outil pourrait par ailleurs fournir un patch sémantique inverse, si cela est possible, pour permettre de facilement revenir en arrière (ou un patch textuel inverse, si le patch sémantique n'est pas inversible).

Une seconde application possible des patches sémantiques serait l'expression d'opérations complexes de refactoring de code. Le développeur pourrait alors décrire un refactoring au moyen d'une combinaison de patches sémantiques, tel que d'ajouter un nouvel argument à toutes les occurrences de définition et d'application d'une fonction particulière. L'outil de refactoring serait alors en charge d'appliquer les patches sémantiques sur l'ensemble du code.

Ces deux applications des patches sémantiques présentent l'avantage de maintenir la connaissance de la sémantique du langage dans des outils dédiés, et non dans l'éditeur de texte qui manipule le projet. Ceci permettrait donc de factoriser la conception de ces outils, pour qu'ils soient ensuite utiliser dans plusieurs éditeurs, sans avoir à redévelopper toute la machinerie souvent complexe des règles métiers du langage.

7. Conclusion

Dans cet article, nous avons présenté les patches sémantiques tels qu'ils sont aujourd'hui utilisés depuis plusieurs années chez OCamlPro. Ils permettent de minimiser le travail de passage des modifications liées au profiling mémoire d'une version d'OCaml à une autre.

Nous pensons que la définition d'un vrai langage de description des patches sémantiques pour OCaml est un travail important, probablement assez difficile, qu'il faudrait faire pour étendre leurs potentielles applications.

Enfin, nous pensons que ce langage de description des patches sémantiques pourrait servir de format d'intéropérabilité entre les outils de développement en OCaml, en particulier pour suggérer des corrections automatiques sur certaines erreur ou pour effectuer du refactoring de code sur un projet entier, sans devoir ré-implanter la connaissance de la sémantique d'OCaml dans chaque éditeur où de tels mécanismes seraient utiles.

Références

[1] J. Lawall. Coccinelle : Reducing the barriers to modularization in a large c code base. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 5–6, New York, NY, USA, 2014. ACM.

[2] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen. Semantic patches considered helpful. *SIGOPS Oper. Syst. Rev.*, 40(3) :90–92, July 2006.

[3] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems : Linguistic Support for Modern Operating Systems*, PLOS '06, New York, NY, USA, 2006. ACM.