Ddddml
Une forme pour les écrire tous

Rapport de stage par

Léo Andrès

15 septembre 2020

Master 2
Fondements de l'Informatique
et Ingénierie du Logiciel

Sous la direction de

Pierre Chambart

Résumé

Ce document est un rapport de mon stage chez OCamlPro, effectué dans le cadre de ma formation en Master 2 FIIL à l'Université Paris-Saclay. J'y présente *dddddml*, un langage où l'on tente d'imposer une forme canonique sur les programmes afin de détecter et d'éviter la duplication de code. J'ai été encadré par Pierre CHAMBART.

Table des matières

1 Contexte du stage

1.1 L'Université Paris-Saclay

Ce stage s'inscrit dans le dans le cadre de ma formation en Master 2 *Fondements de l'Informatique et Ingénierie du Logiciel* à l'Université Paris-Saclay. Il m'était demandé d'effectuer un stage de six mois en entreprise ou bien dans un laboratoire de recherche. Dans une optique de diversification, d'ouverture au monde et ayant choisi pour mes quatre précédents stages un laboratoire de recherche public, j'ai fait cette fois-ci le choix d'un stage en entreprise mais toujours sur un sujet de recherche.

1.2 OCamlPro

L'entreprise Mon stage s'est déroulé au sein d'OCamlPro ; une entreprise qui, comme son nom l'indique, utilise principalement le langage OCaml et est composée d'environ une vingtaine de personnes.

Expertise L'entreprise est spécialisée dans le prototypage et le développement de logiciels en OCaml, les méthodes formelles et le développement de langages de programmation. Quelques exemples sont opam, alt-ergo et flambda. D'autre part, OCamlPro dispense aussi des formations et a contribué à l'écriture de cours en ligne, notamment sur OCaml et Rust.

L'équipe compilation J'ai été encadré par Pierre Chambart qui est directeur de la technologie [1] et ingénieur recherche et développement senior. Il fait partie de l'équipe compilation au sein de laquelle il travaille sur le compilateur OCaml et notamment sur son futur *inliner* : flambda2. Précédemment, l'équipe a publié flambda qui est disponible comme une variante du compilateur OCaml.

Motivation du choix Je souhaitais un sujet de stage portant sur le design et l'implémentation des langages de programmation et où il me serait possible de programmer en OCaml, qui est un langage avec lequel j'ai de fortes affinités et qui est très certainement mon langage favori. OCamlPro m'a donc semblé un choix naturel et évident. J'ai contacté Pierre, que j'avais déjà rencontré par le passé, en lui proposant un sujet que j'avais moi-même écrit. Il m'a donné rendez-vous le lendemain dans les locaux d'OCamlPro et a accepté de m'encadrer. Ainsi débute cette aventure…

1. Ou *Chief Technology Officer* en anglais.

2 Problématique

2.1 Un résultat, plusieurs programmes

Exemple mathématique Il est très facile d'écrire deux expressions mathématiques équivalentes de plusieurs façons différentes. L'expression $2 + 2$ est bien évidemment équivalente à 4 et à $1 + 1 + 1 + 1$.

Exemple informatique Il en va de même pour les programmes informatiques. En OCaml, par exemple, on peut écrire la fonction qui calcule la longueur d'une liste d'au moins trois façons [2] :

Problème Cela pose plusieurs problèmes. Par exemple, dans un programme contenant des millions de lignes et écrit par plusieurs personnes, il peut arriver que l'on écrive plusieurs fois des fonctions équivalentes. On parle de duplication de code. Le programme prend alors plus de place que nécessaire et est plus difficile à maintenir.

Réusinage Lorsqu'un programmeur réalise qu'un programme contient deux fonctions équivalentes, il peut alors procéder à un réusinage [3] afin de n'avoir plus qu'une seule version de cette fonction. Seulement, c'est un processus long et coûteux.

Objectif On aimerait donc pouvoir détecter automatiquement l'équivalence de deux programmes [4] afin d'éviter la duplication de code et de faciliter le réusinage.

2.2 Indécidabilité de l'équivalence extensionnelle

Équivalence extensionnelle La relation entre deux programmes qui *calculent la même chose* est appelée l'*équivalence extensionnelle*.

Indécidabilité Malheureusement, le théorème de Rice[Ric53] implique l'indécidabilité de l'équivalence extensionnelle[MS19]. C'est-à-dire qu'il est impossible d'écrire un programme qui détecte si deux programmes sont équivalents.

2. Et très certainement de beaucoup d'autres.
3. Ou *refactoring* en anglais.
4. Ou plutôt de deux fonctions, ce qui revient au même.

Piste de solution Seulement, ce théorème se place dans le cadre de programmes écrits dans un langage Turing-complet. On peut donc espérer arriver à décider l'équivalence extensionnelle dans un langage qui ne serait pas Turing-complet. La difficulté réside alors dans le fait de conserver un langage qui permette à la fois de calculer des choses *intéressantes* et qui reste suffisamment expressif pour être agréable à utiliser.

2.3 Sujet de stage initial et évolution

L'objectif initial du stage était donc de développer un langage de programmation dans lequel on détecterait l'équivalence extensionnelle au moyen de structures de données telles que les diagrammes de décision binaires[And97] et des généralisations de ceux-ci. Finalement, il a plutôt été question de développer des formes canoniques sur le langage et de forcer le programmeur à les utiliser : il n'est a priori pas aberrant de penser que moins il y a de façons valables d'écrire un programme qui calcule un résultat, plus il est plus facile de décider l'équivalence extensionnelle.

3 Solution apportée

3.1 Le langage dddddml

Le langage développé se nomme `dddddml`. C'est un langage à la ML, dont voici une courte description. Le code source est disponible sur le dépôt git du projet.

3.1.1 Courte description

Indentation Les espaces blancs ne sont pas significatifs, mais on interdit les `trailing whitespaces`, les sauts de ligne inutiles et les espaces consécutifs quand ils ne sont pas en début de ligne. De plus, en début de ligne, on force le nombre d'espaces à être un multiple de deux. Cela ne garantit pas une indentation parfaite, mais permet d'éviter de nombreux problèmes courants tels que la pollution des *diff git* ou les débats sans fin sur le choix de l'utilisation d'espace ou bien de tabulation.

Types de base Deux types de base sont disponibles, le type `unit` et le type *bool*. Le type *unit* contient un unique littéral : `Unit`. Le type *bool* contient deux littéraux : `False` et `True`.

Expression conditionnelle Une instruction de branchement sur les booléens est disponible : `if b then e1 else e2 end`.

Fonction anonyme Il est possible d'écrire des fonctions anonymes : `fun x -> e1`.

Application Il est possible d'appliquer une fonction à une expression de la façon suivante : `f x`.

Liaison Il est possible de lier un identificateur à une expression : `let x = e1 in e2`.

Fonction Il est possible d'utiliser le sucre syntaxique suivant pour définir une fonction à plusieurs arguments : `let f x y z = e1 in e2`.

Priorité Il est possible de changer la priorité de l'application au moyen de parenthèses : `f (g (h x))`.

Types personnalisés Il est possible à l'utilisateur de définir des types qui s'apparentent à des types énumérés de la façon suivante : `let type t = | A | B | C in e`.

Branchement sur les types personnalisés L'utilisateur peut effectuer un branchement sur une expression de type personnalisé ainsi : `let type t = | A | B | C in match e | A -> True | B -> False | C -> False end`.

Sucre syntaxique En réalité, les syntaxes décrites dans les paragraphes *Expression conditionnelle* et *Fonction* ne sont que du sucre syntaxique pour celles décrites dans *Branchement sur les types personnalisés* pour la première et *Liaison* combinée à *Fonction anonyme* pour la seconde.

3.1.2 Grammaire complète

$$
\begin{array}{rcl}
\langle \text{literal} \rangle & ::= & \text{CONID} \\
\langle \text{var\_id} \rangle & ::= & \text{VARID} \\
\langle \text{const} \rangle & ::= & \langle \text{literal} \rangle \\
 & | & \langle \text{var\_id} \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle \text{match\_case} \rangle & ::= & \text{VBAR } \langle \text{literal} \rangle \text{ RARROW } \langle \text{expr} \rangle \\
\\
\langle \text{type\_case} \rangle & ::= & \text{VBAR CONID} \\
\\
\langle \text{expr} \rangle & ::= & \text{LPAR } \langle \text{expr} \rangle \text{ RPAR} \\
& | & \langle \text{const} \rangle \\
& | & \text{FUN } \langle \text{var\_id} \rangle \text{ RARROW } \langle \text{expr} \rangle \\
& | & \text{LET TYPE } \langle \text{var\_id} \rangle \text{ EQ } \langle \text{type\_case} \rangle^* \text{ IN } \langle \text{expr} \rangle \\
& | & \text{LET } \langle \text{var\_id} \rangle \ \langle \text{var\_id} \rangle^* \text{ EQ } \langle \text{expr} \rangle \text{ IN } \langle \text{expr} \rangle \\
& | & \text{IF } \langle \text{expr} \rangle \text{ THEN } \langle \text{expr} \rangle \text{ ELSE } \langle \text{expr} \rangle \text{ END} \\
& | & \text{MATCH } \langle \text{expr} \rangle \ \langle \text{match\_case} \rangle^+ \text{ END} \\
& | & \langle \text{expr} \rangle \ \langle \text{expr} \rangle^+ \\
\\
\langle \text{file} \rangle & ::= & \langle \text{expr} \rangle \text{ EOF}
\end{array}
$$

3.1.3 Système de type

Monomorphisme Le système de type permet uniquement le monomorphisme des fonctions, il ne permet pas le polymorphisme. On peut avoir des variables de type, mais elles ne sont pas quantifiées et seront fixées définitivement à leur première utilisation. Le code `fun x -> x` a bien le type `_t0 -> _t0`, mais si l'on applique cette fonction une fois à une variable de type `unit`, on ne pourra plus l'appliquer qu'à une variable de type `unit`.

Ordre supérieur Le système de type gère parfaitement l'ordre supérieur.

Types récursifs Les types récursifs ne sont pas permis :

```
(let f x = let g x = x in g (g x) in (f f)) True
```

Donne le résultat : `type _t1 is recursive, stop doing this please...`

Bibliothèque générique Le système d'inférence de type, de résolution de contrainte et d'unification est placé dans une bibliothèque séparée, indépendante du système de type de base.

3.2 Vers une forme canonique

Une fois notre langage fixé, il est temps d'imposer des restrictions sur celui-ci, lesquelles seront vérifiées statiquement afin que le programmeur n'ait qu'une

seule façon d'écrire son programme. On donnera ici quelques exemples de telles restrictions. La difficulté est que souvent, les formes canoniques suggérées par la théorie ne sont pas forcément celles qui sont les plus agréables, intuitives ou idiomatiques pour le programmeur. Il s'agit alors de trouver le bon compromis. On veut par exemple empêcher le programmeur de faire des choses *inutiles*.

Variables inutilisées Le code

```
let neg x =
  if x then False else True end
in
Unit
```

donnera l'erreur suivante : `unused variable neg`.

Forçage du sucre syntaxique Dans tous les cas où il est possible d'utiliser du sucre syntaxique, on force l'utilisateur à le faire. Par exemple, le code :

```
let or = fun x -> fun y ->
  if x then True else y end
in
or (or True False) False
```

Donne le résultat : `please use more sugar: you should write` `` `let or x y = ... in ...` `` `instead of` `` `let or = fun x -> fun y -> ... in ...` ``.

Forçage de l'inlining Dans le cas où une variable ou une fonction n'est utilisée qu'une seule fois, on force l'utilisateur à l'inliner. Dans le cas où l'on voudrait rendre possible la création de bibliothèques, on pourrait imaginer l'ajout d'un mot clé `public` qui indiquerait que la fonction est vouée à être exportée et lever cette restriction ainsi que celle décrite dans *Variables inutilisées* :

```
let id x = x in id
```

Donne le résultat : `variable id is used only once, please inline it`.

Tests inutiles Dans le cas où l'utilisateur teste la valeur d'un littéral, on lui signale que c'est inutile puisque qu'on connaît statiquement la branche qui sera prise :

```
if True then True else False end
```

Donne le résultat : `useless if, the condition is constant:` `` `if True then True else False end` ``. Cela fonctionne aussi dans le cas d'un `match`.

Branchement inutile Dans le cas où toutes les branches d'une expression conditionnelle mènent au même résultat, on signale à l'utilisateur que le branchement est inutile :

```
fun x -> if x then x else x end
```

Donne le résultat : `useless if, the two branches are the same: `if x then x else x end``. Cela fonctionne aussi dans le cas d'un `match`.

Branchement inutile bis Dans le cas où le résultat d'un branchement est égal à l'expression testée, il est inutile, par exemple :

```
fun x -> if x then True else False end
```

Donne le résultat : `useless if, it is equivalent to the condition: `if x then True else False end``. Il faudrait alors remplacer toute l'expression `if ... end` par un simple `x`. Cela fonctionne aussi dans le cas d'un `match`.

Réduction de la portée Dans le cas où une variable a une portée [5] plus large que nécessaire, on demande à l'utilisateur de la réduire en déplaçant sa définition. En effet, cela permet de minimiser le nombre de variables accessibles à chaque instruction et il est alors plus facile de raisonner sur le code. Par exemple :

```
let id x = x in
fun x ->
  if id x then x else id (id x) end
```

Donne le résultat : `the scope of the variable id can be reduced`. Il faudrait alors réécrire le code ainsi : `fun x -> let id x = x in if id x then x else id (id x) end`.

Liaisons inutiles Dans le cas où l'utilisateur introduit une liaison inutile, on l'en empêche, par exemple :

```
fun x ->
  let y = x in
  let neg x =
    if x then False else True end
  in
  if neg (neg y) then False else y end
```

Donne le résultat : `variable y is useless, please inline it`.

5. Ou *scope* en anglais.

Application d'une fonction anonyme On empêche l'utilisateur d'appliquer une fonction anonyme, elles ne peuvent qu'être passées en argument à une fonction :

```
(fun x -> if x then True else False end) True
```

Donne le résultat : `you're applying an anonymous function, it's useless, inline the arg and remove that function`.

Propagation d'information Après avoir testé une expression et pris une branche donnée, on sait à l'intérieur de cette branche quelle était la valeur de l'expression. On interdit donc à l'utilisateur de réécrire cette expression au sein de la branche. Par exemple :

```
fun x ->
  if x then x else False end
```

Donne le résultat : `known value for expr `x``. En effet, si l'on a pris la première branche, c'est que la conditionnelle `x` était vraie, on peut donc remplacer `x` par `True` dans la première branche.

Match inutile Dans le cas où l'utilisateur effectue un `match` sur un type à un seul élément, on lui signale que c'est inutile :

```
let type t =
  | A
in
fun x ->
  match x
  | A -> True
  end
```

Donne le résultat : `useless match, it has only one case: `match x | A -> True end``.

3.3 Autres restrictions

Non retenues Certaines restrictions ont été implémentées mais se sont révélées être trop restrictives en pratique, par exemple forcer l'utilisateur à effectuer lui-même une passe de *lambda lifting* rend l'écriture de programme trop laborieuse et la lecture difficile. Il semblerait que forcer le lambda lifting sur certains cas particulier pourrait mener à des résultats intéressants mais je n'ai pas réussi à déterminer précisément lesquels par manque de temps. Le code de certaines de ces restrictions est cependant disponible mais désactivé.

Non implémentées D'autres part, certaines restrictions semblent utiles mais n'ont pas été implémentées. Par exemple, on voudrait éviter l'introduction de fonctions ne servant à rien telles que la fonction `id : let id x = x in id (id True)`. Cependant, il faut le faire de façon précise et selon des critères clairs pour l'utilisateur, ce qui n'a pas été fait par manque de temps là aussi.

3.4 Résultats obtenus

En procédant ainsi, il devient difficile pour l'utilisateur d'écrire de deux façons différentes une fonction calculant un résultat donné. Cependant, c'est toujours possible. L'étape suivante et qui a commencé à être implémentée est celle où l'on détecte que deux fonctions calculent la même chose mais où l'on n'est plus capable d'imposer une forme plutôt qu'une autre. Pour cela, on peut utiliser une généralisation des diagrammes de décision binaires appelée les *discrete decision diagrams* et qui porte sur les fonctions s'appliquant à des types finis. Un début d'implémentation a été réalisé, se basant sur mes travaux effectués précédemment dans [And19]. Seulement, cela n'a pas encore été intégré à *dddddml* car ce n'est pas complètement trivial à effectuer.

3.5 Autres pistes

3.5.1 Isomorphismes de types

Une autre idée serait de détecter les isomorphismes de types comme décrit dans [Ili14]. On pourrait alors brutalement interdire à l'utilisateur d'avoir deux types isomorphes. Seulement, cela nuirait sans doute à la lisibilité et à la compréhension du code. Un système de *vues* comme décrit dans [Wad87] pourrait être mis en place pour palier à ce problème. L'utilisateur n'aurait pas de types isomorphes, mais pourrait avoir des vues ou des alias pour les noms, les constructeurs et les identifiants de fonctions. On pourrait alors essayer d'appliquer un hasconsing à tout le programme et voir quel impact cela aurait sur les performances.

3.5.2 Système d'effets

Notre langage de permet pas d'avoir d'effets de bords. Une solution serait d'ajouter un système d'effets simple où l'on a deux types de flèches : une pour les fonctions pures et une pour les fonctions impures. On forcerait alors l'utilisateur à partitionner autant que possible les fonctions impures et les fonctions pures et l'on relaxerait les contraintes sur les fonctions impures qui sont plus difficiles à analyser statiquement.

3.5.3 Ajout de boucles ou de la récursivité

L'ajout de boucles ou de la récursivité rendrait notre langage Turing-complet, ce que l'on veut éviter. Cependant, il pourrait être possible d'ajouter des constructions natives au langage qui permettent d'émuler des boucles ou la récursivité mais dont on sait qu'elles terminent. C'est par exemple ce qui est fait dans *Gallina*, le langage utilisé dans l'assistant de preuve *Coq*.

3.6 Techniques d'ingénierie du logiciel

Lors du développement de *dddddml*, j'ai tenu à utiliser des techniques modernes d'ingénierie du logiciel. Je tiens à les décrire brièvement ici.

Dépôt git J'ai utilisé une instance Gitea hébergée par mes soins pour héberger mon dépôt git. Gitea est un logiciel libre.

Intégration continue J'ai utilisé build.sr.ht comme système d'intégration continue. C'est un système d'intégration continue libre.

Giteart J'ai utilisé une instance Giteart hébergée par mes soins pour faire le lien entre Gitea et sr.ht.

Déploiement automatique Ainsi, à chaque commit, je vérifie automatiquement l'indentation du code, les tests sont lancés, la documentation est déployée en ligne, la couverture du code par les tests est mesurée et publiée en ligne, la version web de *dddddml* est déployée en ligne, un bot IRC parlant le dddddml est déployé sur un serveur privé, une archive des sources est publiée et finalement, si un tag git est présent, une pull-request est ouverte automatiquement sur opam afin de rendre la dernière version disponible et accessible simplement.

Une bibliothèque de développement de compilateurs Mieux encore, tout ceci est indépendant du langage. J'ai créé une bibliothèque appelée *Complice*. Cette bibliothèque contient des foncteurs, il suffit de lui donner un *lexer*, un *parser* et les transformations à appliquer à l'AST initial et elle génère automatiquement une version web, un bot IRC, un compilateur basique, un REPL etc.

3.7 Autres travaux effectués

D'autre part, j'ai contribué à de nombreux paquets libres maintenus par OCaml-Pro, j'en ai notamment publié une douzaine sur opam. J'ai relevé plusieurs bugs

et corrigés certains d'entre eux sur des paquets n'étant pas maintenus par OCamlPro tels que `dune` ou `bisect_ppx`.

4 Conclusion

Connaissances acquises Ce stage ayant trait à la compilation, il a été pour moi une formidable occasion d'explorer plus en profondeur différents domaines, à travers la lecture d'articles de recherches ou bien de code présent dans d'autres compilateurs : les systèmes de types, les isomorphismes de types, les optimisations des compilateurs etc. De plus, j'ai eu la chance de pouvoir implémenter de nombreux algorithmes et plusieurs applications m'ayant permis de me familiariser avec de nouveaux outils et de nouvelles bibliothèques OCaml. C'était également mon premier stage en entreprise et j'ai ainsi eu l'occasion d'en apprendre plus sur les différents outils développés par OCamlPro.

État du résultat Le résultat obtenu, le langage `dddddml`, même s'il n'est pas réellement utile dans sa forme actuelle, est une base solide et est doté d'une architecture robuste et modulaire. Ainsi, il ne devrait pas être difficile d'implémenter les diverses idées mentionnées précedemment, dans l'optique de tendre vers un langage utile et utilisable.

OCaml D'autre part, une fois de plus, le choix d'OCaml comme langage de programmation s'est révélé extrêment utile, agréable et un soutien précieux. Cela me conforte dans mon point de vue et il reste sans hésitation mon langage favori.

Remerciements

Je souhaite tout d'abord remercier l'ensemble des personnes rencontrées chez OCamlPro, je me suis rapidement senti intégré et à l'aise, même si malheureusement, en raison de la situation actuelle, il ne m'a pas été possible de passer autant de temps que je l'aurai souhaité dans les locaux. Cependant, le peu que j'en ai vu ne m'a laissé entrevoir que des bonnes choses !

D'autre part, ayant eu quelques soucis d'ordre personnel mais aussi beaucoup de rattrapages à passer, j'ai eu la chance d'avoir affaire à des personnes extrêmement compréhensives, sans quoi je n'aurais certainement pas réussi à aller au bout de ce stage, merci à elles, elles se reconnaîtront.

Bien évidemment, je tiens à dire merci à Pierre pour avoir accepté de m'encadrer sur un sujet que j'avais moi-même écrit et m'avoir laissé expérimenter à mon gré diverses choses, cela m'a permis de réaliser un peu mieux en quoi consiste un travail de recherche lorsque l'on ne sait pas forcément où l'on va. Dans les moments de trop grande errance, j'appelais généralement au secours… l'humour et les idées de Pierre m'aidaient alors à retrouver mon chemin. Merci pour m'avoir donné cette chance !

J'ai passé de très bons moments chez OCamlPro et j'espère pouvoir y vivre d'autres aventures…

Enfin, merci à Camille d'avoir été à mes côtés et d'avoir égayé ces derniers mois.

Références

[Ric53] H. G. Rice. « Classes of Recursively Enumerable Sets and Their Decision Problems ». In : *Transactions of the American Mathematical Society* 74.2 (1953), p. 358-366. ISSN : 00029947 (cf. p. 3).

[MS19] Jean-Yves Moyen et Jakob Grue Simonsen. « More Intensional Versions of Rice's Theorem ». In : *Computing with Foresight and Industry*. Sous la dir. de Florin Manea et al. Cham : Springer International Publishing, 2019, p. 217-229. ISBN : 978-3-030-22996-2 (cf. p. 3).

[And97] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams*. Lectures notes from Technical University of Denmark. http://www.cs.utexas.edu/~isil/cs389L/bdd.pdf. 1997 (cf. p. 4).

[And19] Léo Andrès. *Partage d'implémentation, implémentation du partage : une bibliothèque fonctorisée de diagrammes de décision binaires*. https://fs.zapashcanon.fr/pdf/ter-report.pdf. 2019 (cf. p. 10).

[Ili14] Danko Ilik. « Axioms and Decidability for Type Isomorphism in the Presence of Sums ». In : *CSL-LICS '14 Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Sous la dir. d'Henzinger et al. SIGLOG ACM Special Interest Group on Logic and Computation ; EACSL European Association for Computer Science Logic ; IEEE-CS - DATC IEEE Computer Society. Vienna, Austria : ACM, juil. 2014. DOI : 10.1145/2603088.2603115. URL : https://hal.inria.fr/hal-00991147 (cf. p. 10).

[Wad87] P. Wadler. « Views : A Way for Pattern Matching to Cohabit with Data Abstraction ». In : *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany : Association for Computing Machinery, 1987, p. 307-313. ISBN : 0897912152. DOI : 10.1145/41625.41653. URL : https://doi.org/10.1145/41625.41653 (cf. p. 10).