

PSTL : Interface Web Autobill

Fazazi Zeid

Luo Yukai

Brahima Dibassi

23 mars, 2023

Contents

1	Contexte du projet	2
1.1	Qu'est-ce qu'est Autobill ?	2
1.2	Comment on s'inscrit dans ce projet ?	2
1.3	Processus de Conception	3
2	Architecture	4
2.1	Client uniquement	4
2.1.1	Design du client	4
2.1.2	Outils et Technologies utilisées	4
2.1.3	Tâches réalisées	5
2.2	Serveur + Client	6
2.2.1	Schema de Communication	6
2.2.2	Outils et Technologies utilisées	6
2.2.3	Tâches réalisées	7
3	MiniML	8
3.1	Pourquoi MiniML ?	8
3.1.1	Call-By-Push-Value	8
3.2	Description Rapide	8
3.3	Contenu Actuel	8
3.4	Diagramme	9
4	Projections (Rapport Suivant)	10
4.1	MiniML	10
4.2	Serveur	10
4.3	Client	10
4.4	Tests	10
5	Bibliographie	11

1 Contexte du projet

1.1 Qu'est-ce qu'est Autobill ?

Autobill est un projet universitaire soutenu par notre tuteur de projet Hector Suzanne, au sein de l'équipe APR du LIP6, dans la cadre de sa thèse sur l'analyse statique de la consommation mémoire d'un programme. L'analyse statique se réfère au domaine en informatique visant à déterminer des métriques et des comportements à l'exécution d'un programme sans l'exécuter réellement. Les programmes étant écrit dans des langages structurés par une syntaxe précise, en découle alors des sémantiques répondant à différentes problématiques, comme l'évaluation, le typage ou dans le cas de notre projet, l'occupation de ressources par un programme.

On peut définir les ressources comme la quantité de mémoire ou de temps nécessaire à évaluer un programme. Autobill se base sur les travaux de Jan Hoffmann (voir Bibliographie) et l'idée que l'on peut déduire la consommation en ressources depuis des formules arithmétiques. Elles sont issues de l'analyse amortie par méthode de potentiel du coût moyen en ressources, qu'Autobill réalise à chacune des entrées en les traduisant vers un code machine propriétaire décrivant les contraintes logiques du programme.

Le code machine d'Autobill est généré depuis un code en langage **Call-By-Push-Value**. Il utilise paradigme déjà éprouvé, décrit dans les papiers de Paul Blain Lévy (voir Bibliographie), qui utilise une stratégie d'évaluation faisant, entre autres, la distinction entre les expressions de valeurs et les expressions de calculs. On peut ainsi décrire les applications de calculs comme une pile de valeurs / opérandes sur lesquels vont être appliqué le calcul et Autobill utilise ce trait lors de l'analyse de ressource.

1.2 Comment on s'inscrit dans ce projet ?

Le sujet de notre projet STL va donc être de soutenir l'effort de développement en proposant une interface sur le Web permettant la libre manipulation de l'outil Autobill par des tiers à travers un environnement de développement sur navigateur.

Pour le rendre le plus accessible, en entrée, un langage avec une syntaxe similaire à OCaml sera disponible en entrée et pourra être utilisé pour écrire les programmes à tester. Néanmoins, **Autobill** n'acceptant programmes en **Call-By-Push-Value**, il est nécessaire de traduire le code camélien, qui plus vers une stratégie d'évaluation différente (*Call-By-Sharing* et *Call-Py-Push-Value*). Ainsi, un travail sur la compilation est nécessaire, en passant par les étapes de construction d'AST camélien et la traduction de ce dernier en un AST compréhensible par **Autobill**.

1.3 Processus de Conception

Lors de la conception de l'interface les contraintes étaient multiples.

La première était le langage d'implémentation en effet **Autobill** étant développé en **OCaml** il nous était obligatoire de trouver un moyen pour communiquer avec l'application.

La seconde était qu'il fallait développer cette interface en simultané avec **Autobill** et d'ajuster notre travail en fonction des besoins courants de nos encadrants.

Mais la plus importante d'entre elles était le souhait de nos encadrants que l'application soit principalement côté client afin de simplifier son déploiement.

Une fois ses contraintes établies nous avons du tout au long de ce projet effectuer des choix que ce soit en matière de design ou de technologies.

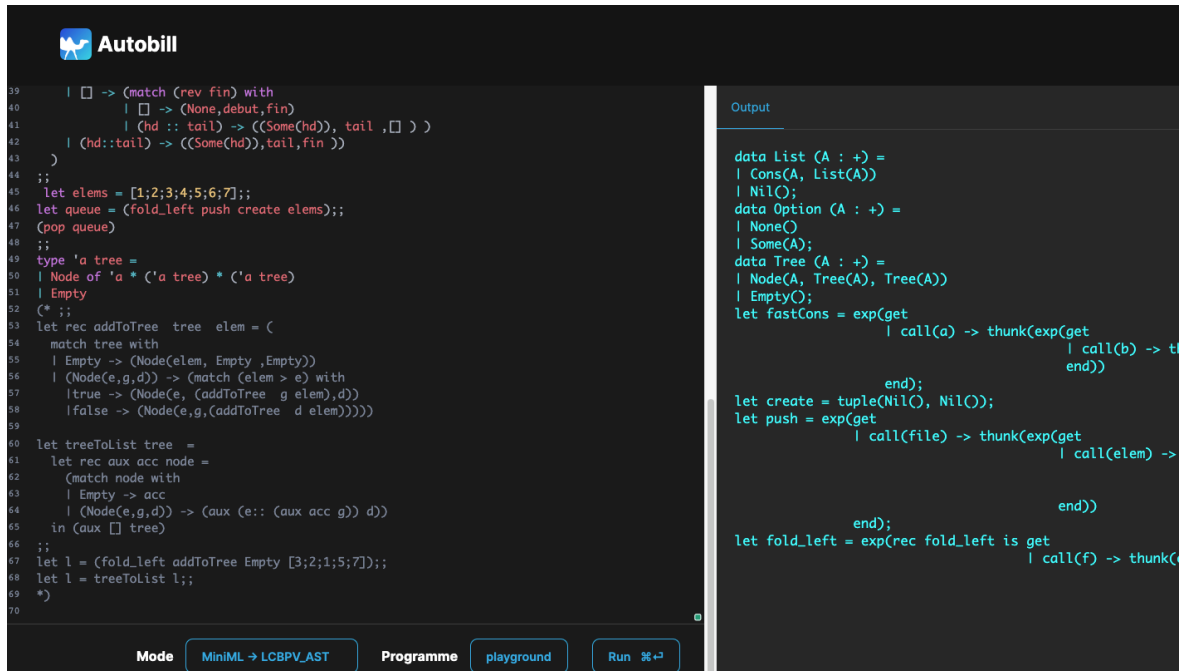
Nous tenons donc à travers ce rapport à mettre en lumière ces décisions tout en décrivant le travail qu'elles ont engendré.

2 Architecture

Dans l'optique de ne pas se restreindre dans un choix de conception, le groupe s'est orienté vers deux structures de projets différentes et indépendantes : l'une fonctionnant avec un client unique, la seconde avec un serveur dédié et un client qui expose ce serveur. L'avantage réside dans le fait que, lors du développement, si un nouvel outil est amené à être utilisé mais ne dispose de compatibilité sur navigateur Web, alors le serveur peut répondre à ce problème. C'est aussi un sujet de comparaison intéressant à présenter par la suite, que ce soit au niveau des performances que du déploiement de ces solutions.

2.1 Client uniquement

2.1.1 Design du client



The screenshot shows the Autobill web application interface. On the left, there is a code editor with a dark background and light-colored text. The code is written in a functional programming style, likely OCaml or Haskell, and includes comments. The code defines a list, a queue, a tree structure, and functions for adding elements to the tree and converting the tree to a list. On the right, there is an 'Output' panel with a light background and dark text. It displays the output of the code, showing the definition of a list, an option type, a tree structure, and functions for fast concatenation, tuple creation, and folding. The interface also includes a 'Mode' dropdown menu with 'MiniML -> LCBPV_AST' selected, a 'Programme' button, a 'playground' button, and a 'Run' button with a keyboard shortcut icon.

```
39 | [] -> (match (rev fin) with
40 | [] -> (None,debut,fin)
41 | (hd :: tail) -> ((Some(hd)), tail ,[] ))
42 | (hd::tail) -> ((Some(hd)),tail,fin ))
43 )
44 ;;
45 let elems = [1;2;3;4;5;6;7];;
46 let queue = (fold_left push create elems);;
47 (pop queue)
48 ;;
49 type 'a tree =
50 | Node of 'a * ('a tree) * ('a tree)
51 | Empty
52 (* ;;
53 let rec addToTree tree elem = (
54 match tree with
55 | Empty -> (Node(elem, Empty ,Empty))
56 | (Node(e,g,d)) -> (match (elem > e) with
57 | true -> (Node(e, (addToTree g elem),d))
58 | false -> (Node(e,g,(addToTree d elem))))))
59
60 let treeToList tree =
61 let rec aux acc node =
62 (match node with
63 | Empty -> acc
64 | (Node(e,g,d)) -> (aux (e:: (aux acc g) ) d))
65 in (aux [] tree)
66 ;;
67 let l = (fold_left addToTree Empty [3;2;1;5;7]);;
68 let l = treeToList l;;
69 *)
70
```

```
data List (A : +) =
| Cons(A, List(A))
| Nil();
data Option (A : +) =
| None()
| Some(A);
data Tree (A : +) =
| Node(A, Tree(A), Tree(A))
| Empty();
let fastCons = exp(get
| call(a) -> thunk(exp(get
| call(b) -> t
end))
end);
let create = tuple(Nil(), Nil());
let push = exp(get
| call(file) -> thunk(exp(get
| call(elem) ->
end))
end);
let fold_left = exp(rec fold_left is get
| call(f) -> thunk(
```

2.1.2 Outils et Technologies utilisées

- **HTML / CSS / Javascript** : Il s'agit de la suite de langages principaux permettant de bâtir l'interface Web souhaitée. On a ainsi la main sur la structure de la page à l'aide des balises HTML, du style souhaité pour l'éditeur de code avec le CSS et on vient apporter l'interactivité et les fonctionnalités en les programmant avec Javascript, complété par la librairie React.
- **React.js** : React s'ajoute par-dessus la stack technique décrite plus haut pour proposer une expérience de programmation orientée composante sur le Web.

C'est une librairie Javascript permettant de construire des applications web complexes tournant autour de composants / éléments possédant un état que l'on peut imbriquer entre eux pour former notre interface utilisateur et leur programmer des comportements et des fonctionnalités précises, sans se soucier de la manipulation du DOM de la page Web.

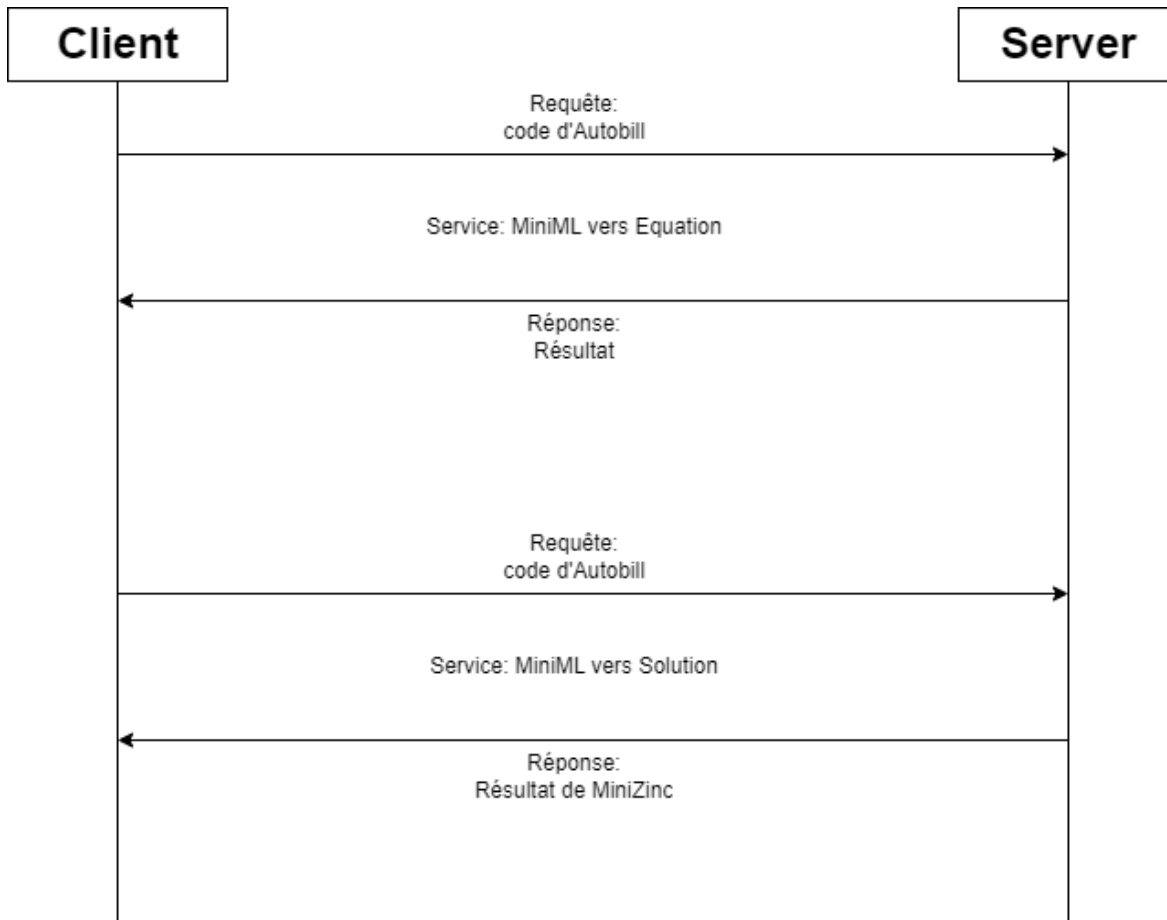
- **CodeMirror** : C'est une librairie Javascript permettant d'intégrer un éditeur de code puissant, incluant le support de la coloration syntaxique, de l'auto-complétion ou encore le surlignage d'erreurs. Les fonctionnalités de l'éditeur sont grandement extensives et permettant même la compatibilité avec un langage de programmation personnalité comme **MiniML**. Enfin, CodeMirror est disponible sous licence MIT.
- **OCaml + Js_of_OCaml** : Afin de manipuler la librairie d'**Autobill**, il est nécessaire de passer par du côté OCaml pour traiter le code en entrée et en sortir des équations à résoudre ou des résultats d'interprétations. Pour faire le pont entre Javascript et OCaml, on utilise Js_of_OCaml, une librairie contenant, entre autres, un compilateur qui transpile du bytecode OCaml en Javascript et propose une grande variété de primitive et de type pour manipuler des éléments Javascript depuis OCaml

2.1.3 Tâches réalisées

- Intégration d'une IDE similaire aux Playground de OCaml et Rescript
- Implémentation d'un éditeur de code supportant la syntaxe de **MiniML**
- Liaison entre le code Javascript et OCaml à l'aide de Js_of_OCaml
- Implémentation de plusieurs modes d'interprétation du code **MiniML** :
 - Vers AST
 - Vers AST de **Call-By-Push-Value**
 - Vers Equation
 - Vers code Machine **Autobill**
- Remontée d'erreurs et affichage dynamique sur l'interface
- Implémentation du solveur d'équations MiniZinc côté client
- Solveur (Web Assembly)

2.2 Serveur + Client

2.2.1 Schema de Communication



2.2.2 Outils et Technologies utilisées

2.2.2.1 Coté Client

- HTML / CSS / Javascript
- React.js
- CodeMirror
- OCaml + Js_of_OCaml

2.2.2.2 Coté Serveur

- **NodeJS**: NodeJS permet une gestion asynchrone des opérations entrantes, ce qui permet d'avoir une grande efficacité et une utilisation optimale des ressources. En outre, NodeJS est également connu pour son excellent support de la gestion des entrées/sorties et du traitement de données en temps réel.

Enfin, la grande quantité de packages disponible sur NPM (le gestionnaire de packages de Node Js) permet de gagner beaucoup de temps de développement et de faciliter notre tâche.

2.2.3 Tâches réalisées

- Intégration d'une IDE similaire aux Playground de OCaml et Rescript
- Implémentation d'un éditeur de code supportant la syntaxe de **MiniML**
- Liaison entre le code Javascript et OCaml à l'aide de Js_of_OCaml
- Implémentation de plusieurs modes d'interprétation du code **MiniML** :
 - Vers Equation
- Implémentation du solveur d'équations MiniZinc côté client
- Solveur

3 MiniML

3.1 Pourquoi MiniML ?

MiniML emerge de la volonté de créer un langage fonctionnel simple, accessible et sans effets de bord pour les utilisateurs d'autobill car celui-ci requiert une connaissance approfondie de la théorie autour des différentes sémantiques d'évaluation a travers le paradigme **Call-By-Push-Value**.

3.1.1 Call-By-Push-Value

Le paradigme de traitement de langage **Call-By-Push-Value** utilisé par autobill permet à l'aide d'une seule sémantique de traiter deux types de stratégies d'évaluation différentes **Call By Value** utilisée par **OCaml** et **Call By Name** utilisée par **Haskell** pour mettre en place l'évaluation *Lazy*.

Pour permettre cette double compatible **Call-By-Push-Value** effectue une profonde distinction entre les calculs et les valeurs.

La différenciation entre ses deux types de stratégies s'effectue lors de la traduction depuis le langage d'origine.

3.2 Description Rapide

MiniML dans ce projet dispose d'une implémentation écrite en **OCaml**.

MiniML possède deux types de base (Integer et Boolean).

Il est possible de créé de nouveaux types à partir de ceux-ci.

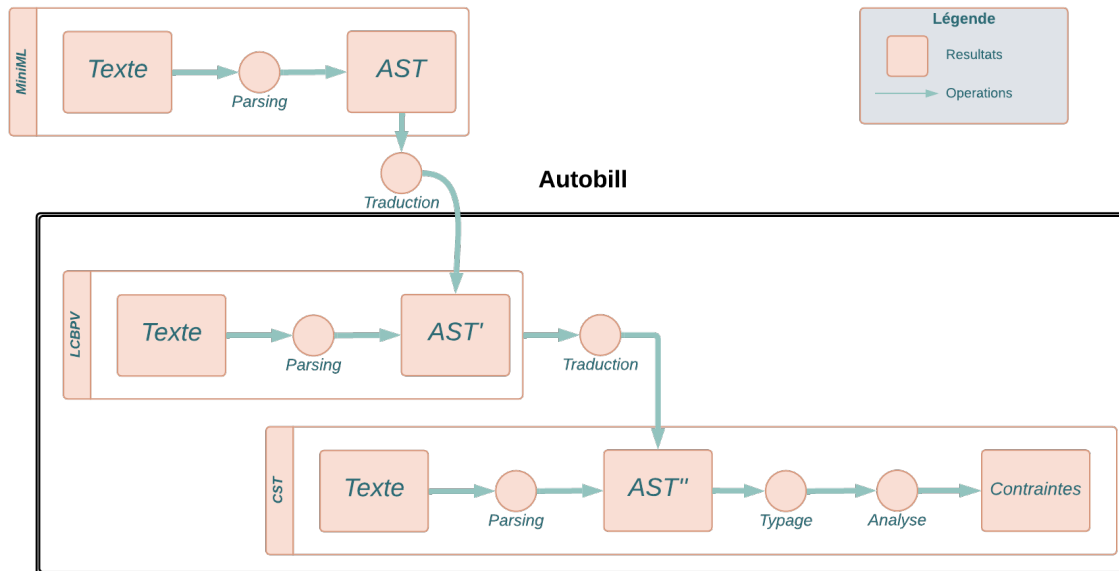
La syntaxe de MiniML est très proche de celle d'OCaml et parfaitement compatible avec un parseur OCaml

3.3 Contenu Actuel

- Listes
- Files
- Fonction Recusives
- Operateurs de Bases
- Construction de Types
- Variables Globales/Locales

3.4 Diagramme

Connection MiniML-AutoBill



4 Projections (Rapport Suivant)

4.1 MiniML

- Ajout de sucre syntaxique.
- Ajout d'une librairie standard.
- Specification complete du langage.
- Bibliothèque de tests sous la forme de structure de données complexes

4.2 Serveur

- Affichage des erreurs
- Réalisation des autres services pour MiniML
- Réalisation de génération de solution depuis le code Autobill

4.3 Client

- Retouches esthétiques
- Affichage des erreurs sur plusieurs lignes
- Couverture d'erreurs à traiter la plus grande possible, afin d'éviter les blocages du client
- "Benchmark" la résolution d'équations plus complexes avec le MiniZinc client
- Proposer des programmes d'exemples à lancer, demandant des lourdes allocations mémoires.

4.4 Tests

- Comparaison d'architectures Full-Client vs Client-Serveur
- Comparaison **RAML** vs **Autobill**

5 Bibliographie

- Will Kurt. 2018. Get Programming with Haskell. Simon and Schuster. Chapitre 5,7
- Pierce, Benjamin C. Types and Programming Languages. MIT Press, 2002
- Levy, Paul Blain. “Call-by-Push-Value: A Subsuming Paradigm.” Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. 228–243
- Winskel, Glynn. The Formal Semantics of Programming Languages: an Introduction. Cambridge (Mass.) London: MIT Press, 1993
- Compilers: Principles, Techniques, and Tools. 2nd ed. Boston (Mass.) San Francisco (Calif.) New York [etc: Pearson Addison Wesley, 2007]
- Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. Real World OCaml. O’Reilly Media.
- Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. Proceedings of the ACM on Programming Languages 1, 1-29
- Hoffmann, Jan, and Steffen Jost. “Two Decades of Automatic Amortized Resource Analysis.” Mathematical structures in computer science 32.6 (2022): 729–759
- Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. Proceedings of the ACM on Programming Languages 3, ICFP (2019)
- Xavier Leroy. 2022 OCaml library. OCaml Lazy Doc. Retrieved February 20, 2023 **Link**
- Haskell - Wikibooks, open books for an open world. Doc Haskell. Retrieved February 17, 2023 **Link**