

# PSTL : Interface Web Autobill

Fazazi Zeid

Luo Yukai

Brahima Dibassi

23 mars, 2023

## Contents

<b>1</b>	<b>Contexte du projet</b>	<b>2</b>
1.1	Qu'est-ce qu'est Autobill ? . . . . .	2
1.2	Comment on s'inscrit dans ce projet ? . . . . .	2
<b>2</b>	<b>MiniML</b>	<b>3</b>
2.1	Call-By-Push-Value . . . . .	3
2.2	Objectifs du langage . . . . .	3
2.3	Description Rapide . . . . .	3
2.4	Contenu Actuel . . . . .	3
2.5	Processus de Conception . . . . .	4
2.6	Diagramme . . . . .	4
<b>3</b>	<b>Architecture</b>	<b>4</b>
3.1	Client uniquement . . . . .	5
3.1.1	Design du client . . . . .	5
3.1.2	Outils et Technologies utilisées . . . . .	5
3.1.3	Tâches réalisées . . . . .	6
3.2	Serveur + Client . . . . .	7
3.2.1	Schema de Communication . . . . .	7
3.2.2	Outils et Technologies utilisées . . . . .	7
3.2.3	Tâches réalisées . . . . .	8
<b>4</b>	<b>Projections (Rapport Suivant)</b>	<b>8</b>
4.1	MiniML . . . . .	8
4.2	Serveur . . . . .	8
4.3	Client . . . . .	8
4.4	Tests . . . . .	8
<b>5</b>	<b>Bibliographie</b>	<b>9</b>

# 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, que Autobill réalise à chacune des entrées en les traduisant vers un code machine propriétaire décrivant les contraintes logiques du programme.

## 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.

Notre approche vise donc à faciliter l'utilisation d'**Autobill** avec une interface Web qui prendrait la forme d'un "mini" environnement de développement, avec un éditeur de code et une sortie standard sur le côté. Aussi, pour le rendre le plus accessible, en entrée, un langage avec un syntaxe similaire à Ocaml sera disponible en entrée et pourra être utilisé pour écrire les programmes à tester.

Néanmoins, le langage accepté d'**Autobill** étant **Call-By-Push-Value**, il est nécessaire de pouvoir traduire le langage camélien pour permettre l'évaluation des contraintes et de l'allocation mémoire du programme. Ainsi, un travail sur la compilation d'un langage à un autre va avoir lieu, passant par les étapes de construction d'AST camélien et par la traduction de ce dernier en un AST compréhensible par **Autobill**.

La charge de travail pour notre trinôme va donc se diviser autour de trois axes principaux : le traitement des entrées en **MiniML** pour les convertir en **Call-By-Push-Value** et le développement du client et du serveur Web.

## 2 MiniML

Avant de pouvoir entrer dans le détail au sujet de MiniML une courte introduction au principe de **Call-By-Push-Value** est requise

### 2.1 Call-By-Push-Value

Le paradigme de traitement de langage **Call-By-Push-Value** est utilisé par **autobill** avec un objectif principal permettre avec une seule sémantique de traiter deux types de stratégies d'évaluation différentes **Call By Value** utilisée par **Ocaml** par exemple et **Call By Name** ou utilisée par **Haskell** la différenciation entre ses deux types de stratégies s'effectue lors de la traduction depuis le langage d'origine.

Pour ce faire **Call-By-Push-Value** effectue une profonde distinction entre les calculs et les valeurs

### 2.2 Objectifs du langage

Pour utiliser ce paradigme **MiniML** a été mis en place dans le cadre de ce PSTL comme modeste subset d'ocaml dont l'objectif est double, faciliter l'utilisation d'**autobill** en offrant une abstraction simple et accessible de **Call-By-Push-Value**, et permettre de simplifier les comparaisons avec d'autres analyseurs. En effet les programmes MiniML étant valides pour tout autre analyseur recevant ocaml en entrée.

### 2.3 Description Rapide

**MiniML** possède deux types de base (Int et Boolean).

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

MiniML pour l'instant est purement fonctionnelle et donc sans noyau impératif.

### 2.4 Contenu Actuel

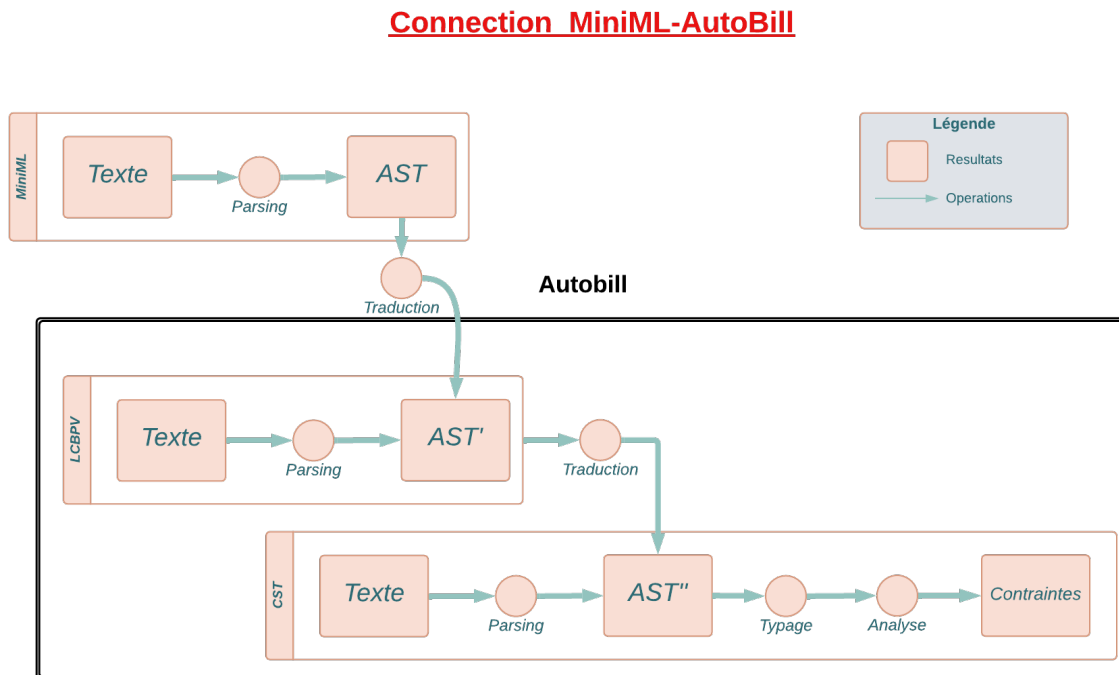
- Listes
- Files
- Fonction Recursives
- Opérateurs de Bases
- Types Constructeurs
- Variable Globales/Locales

## 2.5 Processus de Conception

Lors de la conception de MiniML les contraintes étaient multiples. La plus forte d'entre elles était la nécessité d'être soumis à un minimum de dépendance possible afin de permettre la réutilisation de l'effort de développement dans les deux architectures décrites et mise en place lors du PSTL et la seconde se trouvait au niveau de la traduction de MiniML vers **Call-By-Push-Value** le principe de stratégie d'évaluation étant tout nouveau pour nous.

Une fois ses contraintes établies nous avons décidé d'utiliser ocaml avec comme unique dépendance **menhir**, ce choix nous a permis d'effectuer la traduction d'AST *MiniML* vers AST *Call-By-Push-Value* guidée pas à pas par nos tuteurs car le langage d'implémentation est identique entre MiniML et Autobill

## 2.6 Diagramme



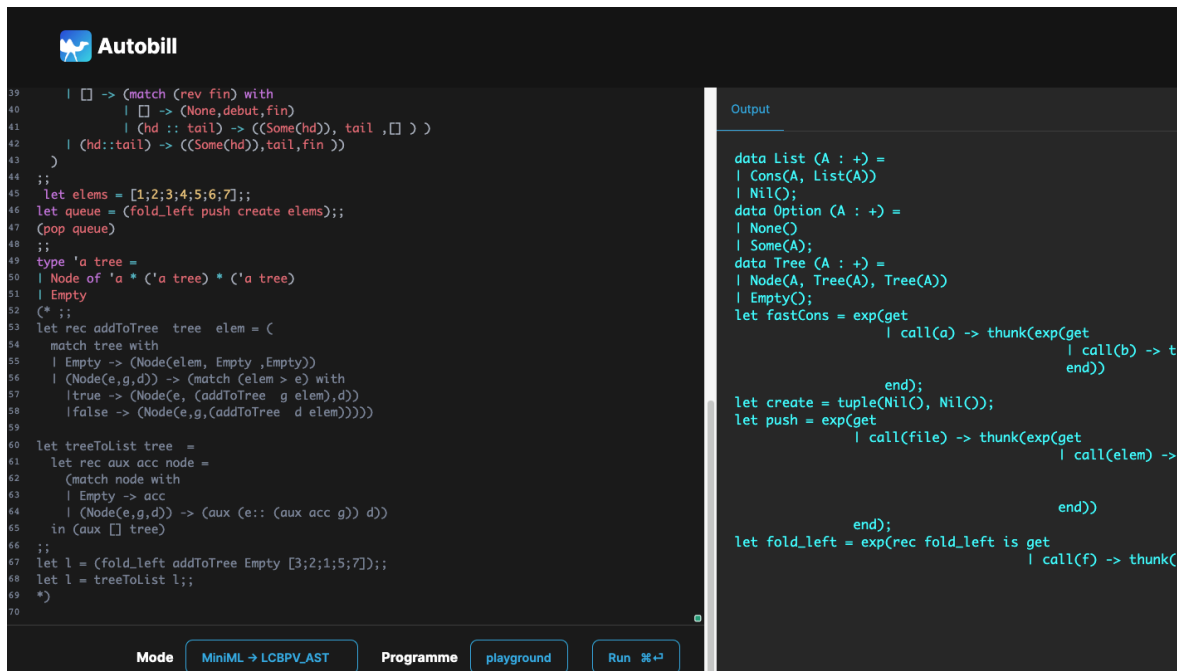
## 3 Architecture

Dans l'optique de ne pas se restreindre dans un choix de conception, le groupe s'est orienté vers deux structures de projet 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, que ce soit au niveau des performances que du déploiement de ces solutions.

## 3.1 Client uniquement

### 3.1.1 Design du client



The screenshot shows the Autobill code editor interface. On the left, the code editor displays MiniML code for a list and tree structure. On the right, the output window shows the compiled code.

```

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(

```

### 3.1.2 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 souhaitée 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 composant 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 la 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.

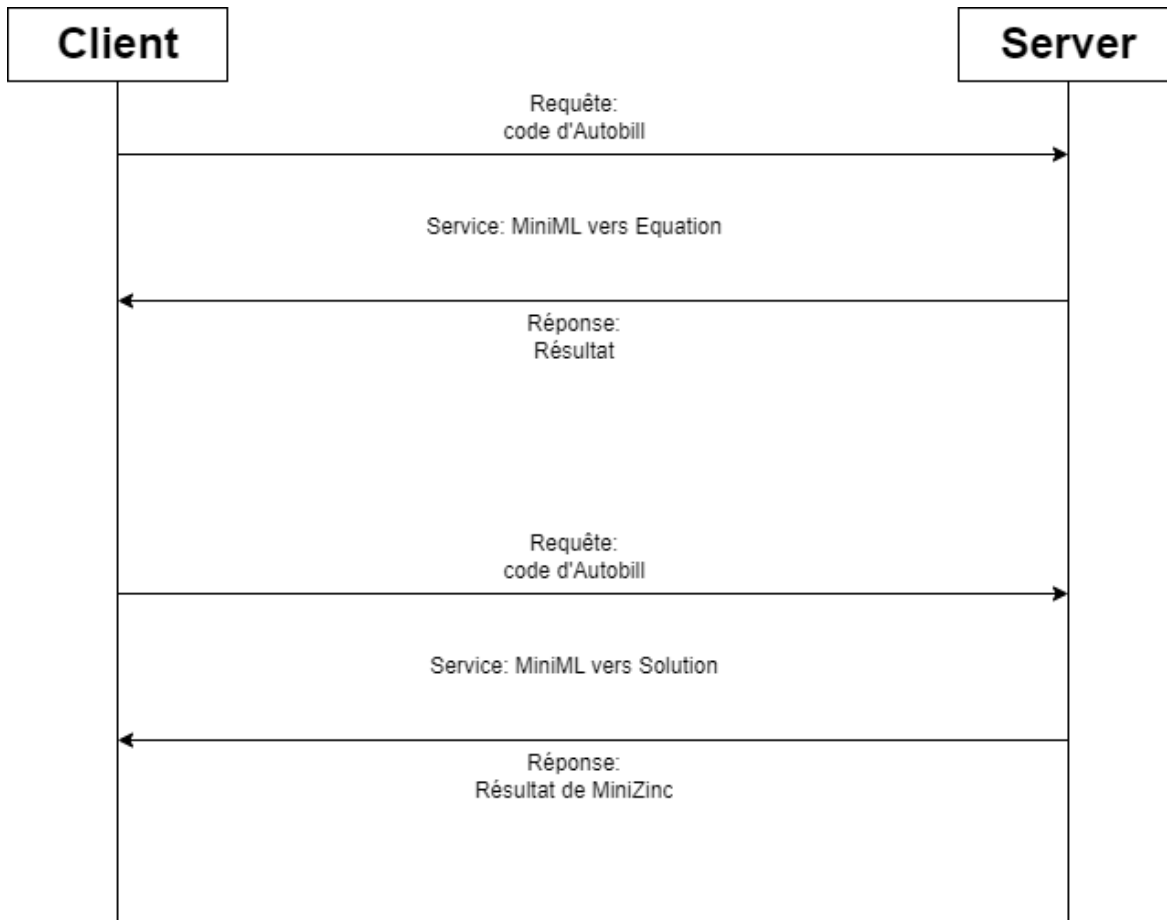
- **Ocaml + Js\_of\_ocaml** : Afin de manipuler la librairie d'**Autobill**, il est nécessaire de passer par du côté Ocaml pour traiter le code en entrée et en sortir des équations à résoudre ou des résultats d'interprétations. Pour faire le pont entre Javascript et Ocaml, on utilise Js\_of\_ocaml, une librairie contenant, entre autres, un compilateur qui transpile du bytecode Ocaml en Javascript et propose une grande variété primitives et de type pour manipuler des éléments Javascript depuis Ocaml

### 3.1.3 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)

## 3.2 Serveur + Client

### 3.2.1 Schema de Communication



### 3.2.2 Outils et Technologies utilisées

#### 3.2.2.1 Coté Client

- HTML / CSS / Javascript
- React.js
- CodeMirror
- Ocaml + Js\_of\_ocaml

#### 3.2.2.2 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 disponibles sur NPM (le gestionnaire de packages de NodeJS) permet de gagner beaucoup de temps de développement et de faciliter notre tâches.

### 3.2.3 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

## 4 Projections (Rapport Suivant)

### 4.1 MiniML

- Ajout de sucre syntaxique.
- Ajout d'une librairie standard.
- Specification complète 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 modalities. Proceedings of the ACM on Programming Languages 3, ICFP (2019)
- Xavier Leroy. 2022 OCaml library. Ocaml Lazy Doc. Retrieved February 20, 2023 from <https://v2.ocaml.org/api/index.html>
- Haskell - Wikibooks, open books for an open world. Doc Haskell. Retrieved February 17, 2023 from <https://en.wikibooks.org/wiki/Haskell>