

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
2	Avancement	3
2.1	MiniML	3
2.1.1	Call-By-Push-Value	3
2.1.2	Objectifs du langage	3
2.1.3	Description Rapide	3
2.1.4	Contenu Actuel	3
2.1.5	Processus de Conception	4
2.1.6	Diagramme	4
2.2	Architecture Full Client	5
2.2.1	Design du client	5
2.2.2	Outils et Technologies utilisées	5
2.2.3	Tâches réalisées	6
2.3	Architecture Serveur + Client	7
2.3.1	Schema de Communication	7
2.3.2	Outils et Technologies utilisées	7
2.3.3	Tâches réalisées	8
3	Projections (Rapport Suivant)	8
3.1	MiniML	8
3.2	Serveur	8
3.3	Client	8
3.4	Tests	8
4	Bibliographie	9

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. Sur la base d'un langage de programmation fonctionnel-impératif nommé **Call-By-Push-Value**, **Autobill** permet de déduire des équations sur les contraintes mémoires du programme qu'on lui passe en entrée. Les équations prennent la forme de formules logiques que l'on peut résoudre grâce à des solveurs externes et en tirer les bornes minimums et variables permettant de satisfaire les contraintes.

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. En effet, il n'est pour l'heure uniquement utilisable via l'invite de commandes, en ayant au préalable cloner le repertoire GitLab où il est hébergé, et suivi les étapes d'installation, nécessitant des utilitaires de paquets comme opam et dune.

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. Plusieurs modes d'évaluation seront disponibles, comme la possibilité d'interpréter ou d'afficher l'occupation mémoire minimum du programme en entrée.

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. Pour ces deux derniers, un effort sera mis à comparer notamment les choix de conception orientés soit vers une architecture tout-client et une architecture client-serveur, les avantages et inconvénients de chacun.

2 Avancement

2.1 MiniML

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

2.1.1 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

2.1.2 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.

2.1.3 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.

2.1.4 Contenu Actuel

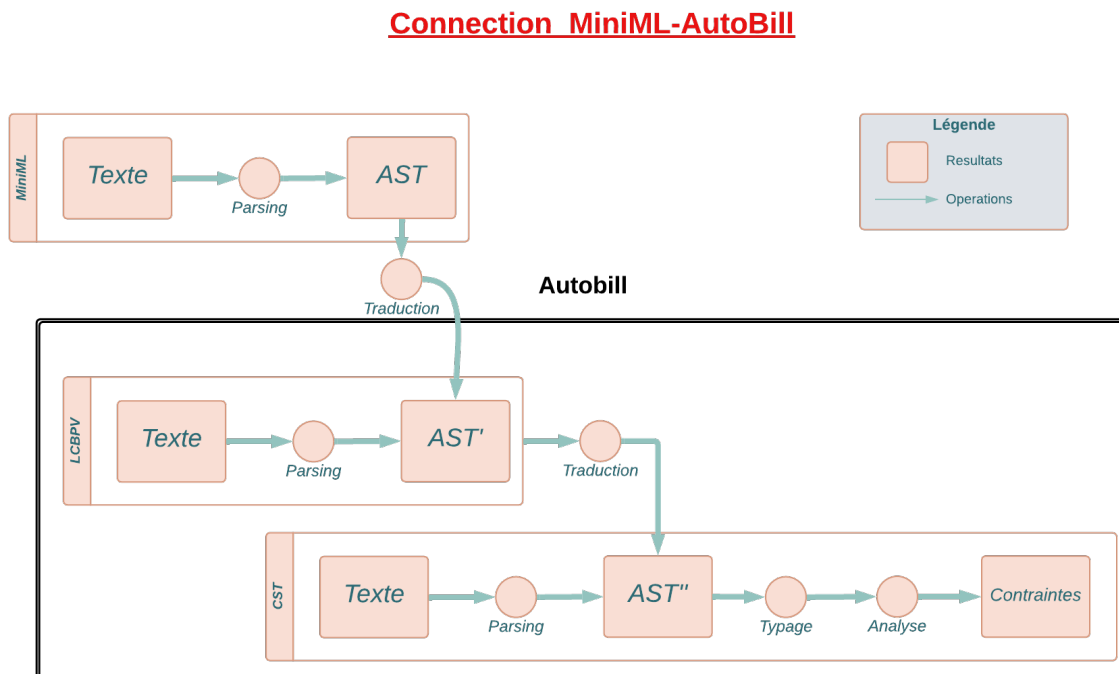
- Listes
- Files
- Fonction Recursives
- Opérateurs de Bases
- Types Constructeurs
- Variable Globales/Locales

2.1.5 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.

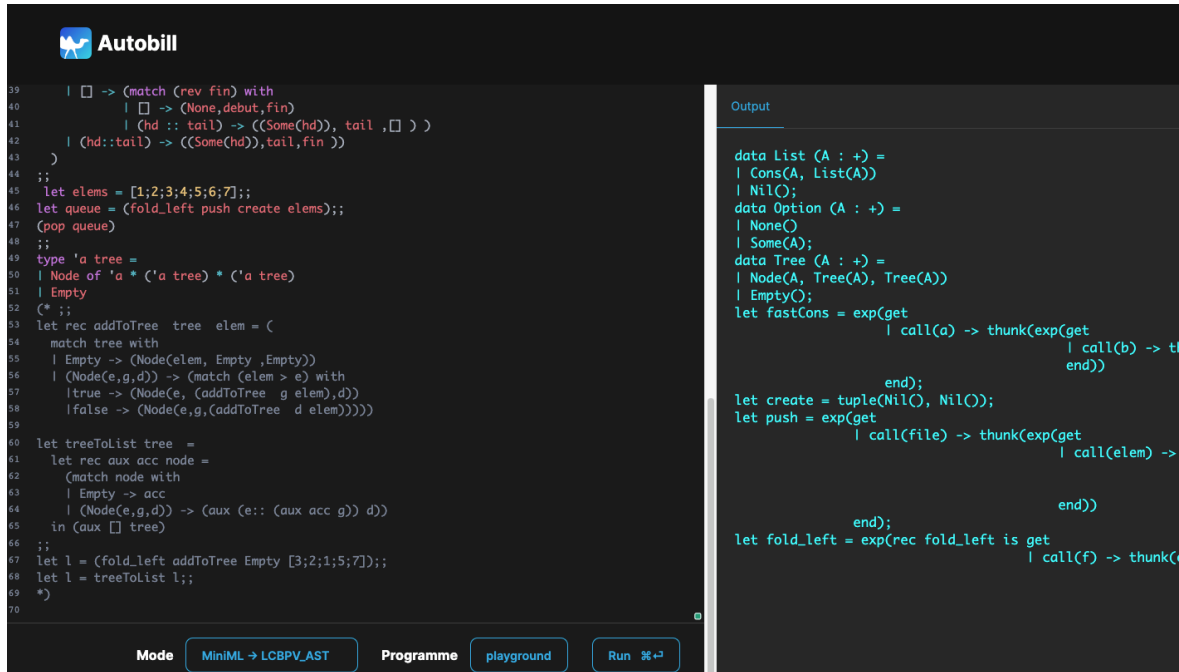
Une fois ses contraintes établies nous avons décidé d'utiliser ocaml avec comme unique dépendance **menhir**, ce choix nous a permis d'effectuer la traduction d'AST *MiniML* vers AST *Call-By-Push-Value* guidée pas à pas par nos tuteurs car le langage d'implémentation est identique entre MiniML et Autobill

2.1.6 Diagramme



2.2 Architecture Full Client

2.2.1 Design du client



The screenshot shows the Autobill web application interface. On the left, there is a code editor with a dark theme, displaying a MiniML program. The code defines a list, a queue, a tree structure, and functions for adding elements to the tree and converting it back to a list. On the right, there is an 'Output' panel showing the compiled code for the same program. At the bottom, there are buttons for 'Mode' (set to 'MiniML → LCBPV_AST'), 'Programme', 'playground', and 'Run'.

```
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(
```

2.2.2 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.

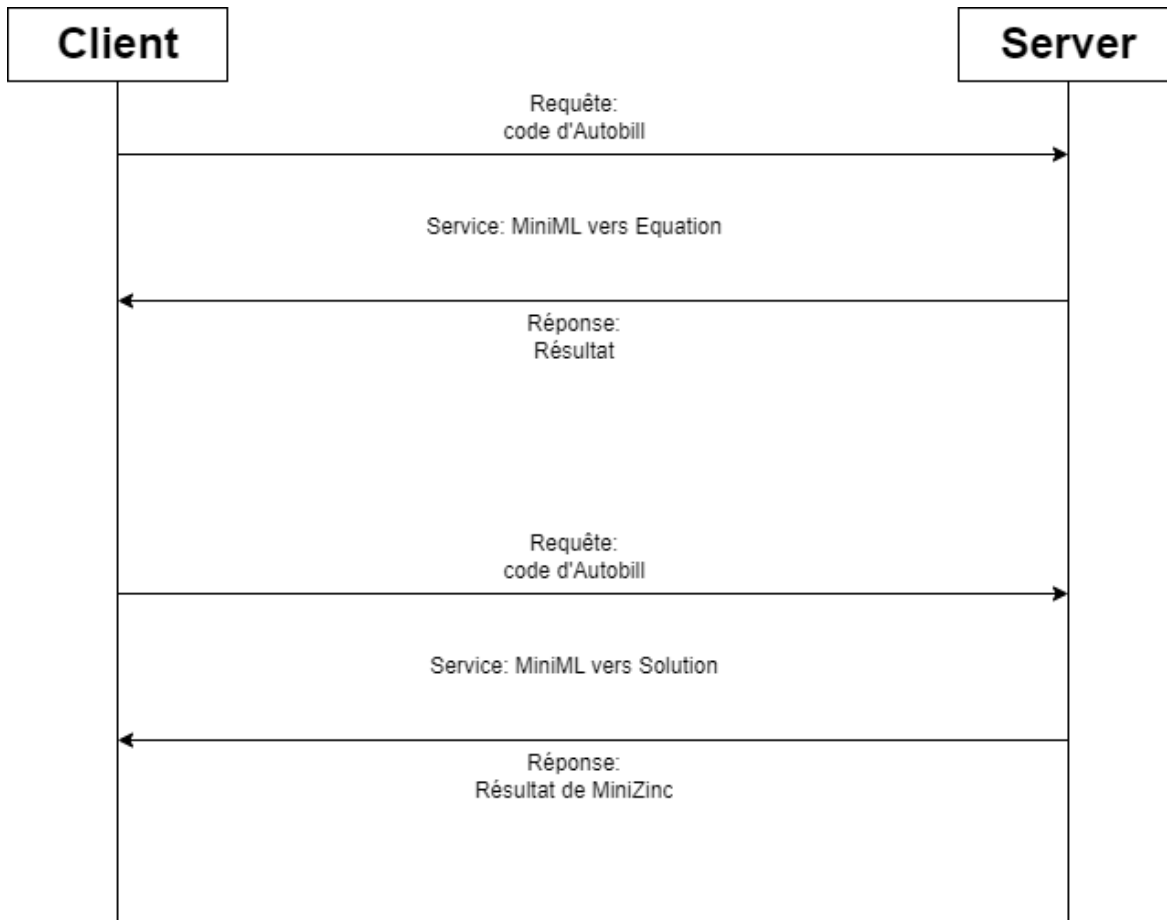
- **Ocaml + Js_of_ocaml** : Afin de manipuler la librairie d'**Autobill**, il est nécessaire de passer par du côté Ocaml pour traiter le code en entrée et en sortir des équations à résoudre ou des résultats d'interprétations. Pour faire le pont entre Javascript et Ocaml, on utilise Js_of_ocaml, une librairie 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

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 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)

2.3 Architecture Serveur + Client

2.3.1 Schema de Communication



2.3.2 Outils et Technologies utilisées

2.3.2.1 Coté Client

- HTML / CSS / Javascript
- React.js
- CodeMirror
- Ocaml + Js_of_ocaml

2.3.2.2 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.

2.3.3 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 Projections (Rapport Suivant)

3.1 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

3.2 Serveur

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

3.3 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.

3.4 Tests

- Comparaison d'architectures Full-Client vs Client-Serveur

- Comparaison **RAML** vs **Autobill**

4 **Bibliographie**

- Reprendre le carnet de bord