PSTL : Interface Web Autobill

Pré-Rapport

Fazazi Zeid Luo Yukai Dibassi Brahima

Encadrants : Hector Suzanne, Emmanuel Chailloux

Table des matières

1 Contexte du projet

1.1 Historique et définitions

Autobill est un projet universitaire développé par notre tuteur de projet Hector Suzanne, au sein de l'équipe APR du LIP6, dans le cadre de sa thèse sur l'analyse statique de la consommation mémoire d'un programme.

L'analyse statique est un domaine de l'informatique qui consiste à mesureer et détecter automatiquement les comportements ou erreurs dans un programme en examinant son code source. Pour effectuer cette analyse sur un langage de programmation donné, il est possible de définir des règles d'évaluation et de typage. Dans notre situation spécifique, nous sommes particulièrement intéressés par l'occupation de la mémoire d'un programme.

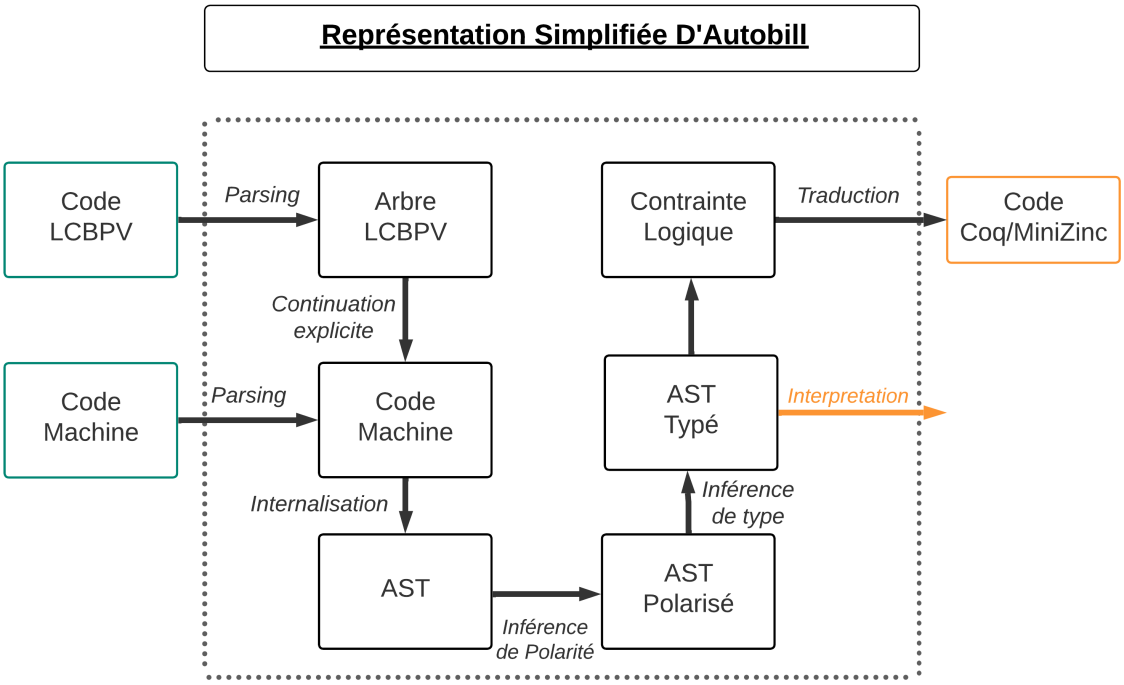
Historiquement, ce sujet de recherche a été plusieurs fois abordé dans divers travaux scientifiques, parmi eux, ceux de Jan Hoffmann sur l'analyse de consommation de ressources automatisé (AARA) [1]. Des solutions se basant sur ces théories existent, comme RAML [2] (Resource Aware ML), un langage *à la ML* permettant ce type d'analyse, créé par Jan Hoffman et Stephen Jost.

1.2 Qu'est-ce qu'Autobill ?

La proposition d'Hector Suzanne avec Autobill se différencie par un niveau d'analyse plus précis sur les fermetures et les arguments fonctionnels d'un programme. D'abord, Autobill prend en entrée des programmes écrits soit en modèle machine propre à Autobill, soit en **Call-By-Push-Value** (CBPV), avec ou sans continuation explicite.

C'est un langage qui utilise un paradigme déjà éprouvé, décrit dans la thèse de Paul Blain Lévy [3]. CBPV utilise une pile pour stocker les valeurs et les fonctions manipulées dans le programme. Ainsi, on peut suivre de manière explicite les quantités de mémoire pour chaque valeur introduite/éliminée ou fonction appelée/terminée. Aussi, le langage permet d'exprimer clairement les stratégies d'évaluation utilisées dans le code source : on fixe quand les évaluations se déroulent, afin de mieux prédire la consommation de mémoire à chaque étape du programme.

À partir d'une entrée en CBPV, Autobill traduit le programme en un code machine avec continuation, exprimant explicitement les contraintes de taille qui s'appliquent sur l'entrée. Il l'internalise, c'est à dire construit l'arbre syntaxique abstrait (AST) de ce programme. Ensuite, Autobill infère dans l'AST le typage de ses expressions ainsi que leurs polarités. Enfin, il en tire en sortie les contraintes dans des formats d'entrées supportés par différents outils de recherche opérationnelles et assistants de preuve, comme MiniZinc ou Coq, afin de prouver des propriétés de complexité temporelle ou spatiale.



1.3 Objectifs du projet

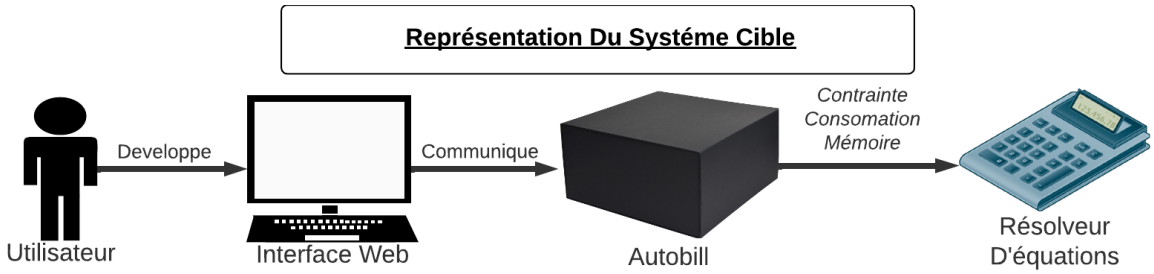
Notre démarche se rapproche de celle de RAML [2] dans leur site officiel.

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 utilisateurs à travers un environnement de développement sur navigateur.

On souhaite aussi faciliter l'utilisation de l'outil avec un langage fonctionnel pur en entrée plus accessible, un **MiniML**. Cela nous contraint donc à adapter cette nouvelle entrée pour qu'elle soit compatible avec Autobill. Enfin, on se charge aussi de traiter les différentes sorties standards et d'erreurs d'Autobill, notamment les expressions de contraintes, afin de les passer à des solveurs externes, en tirer des preuves de complexité et les afficher directement sur le client Web.

Notre charge de travail doit se diviser en plusieurs tâches principales :

- L'implémentation du langage MiniML et sa traduction vers LCBPV
- La mise en place d'une interface Web
- La mise en relation entre l'interface Web et la machine Autobill
- Le traitement des contraintes d'Autobill par un solveur externe
- Les tests de performances et comparaisons avec les solutions existantes



1.4 Processus de conception

Lors de la conception de l'interface, les contraintes étaient multiples. La première était l'interopérabilité des technologies du projet. En effet **Autobill** étant développé en **OCaml**, il était nécessaire de trouver des moyens pour l'adapter à un environnement Web.La seconde était qu'il fallait développer cette interface en simultané avec **Autobill** et 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 dans les infrastructures de la faculté.

Une fois ces contraintes établies, nous avons dû,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 Interface web

Dans l'optique de ne pas se restreindre dans l'utilisation d'outils notamment au niveau du résolveur de contraintes, 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.

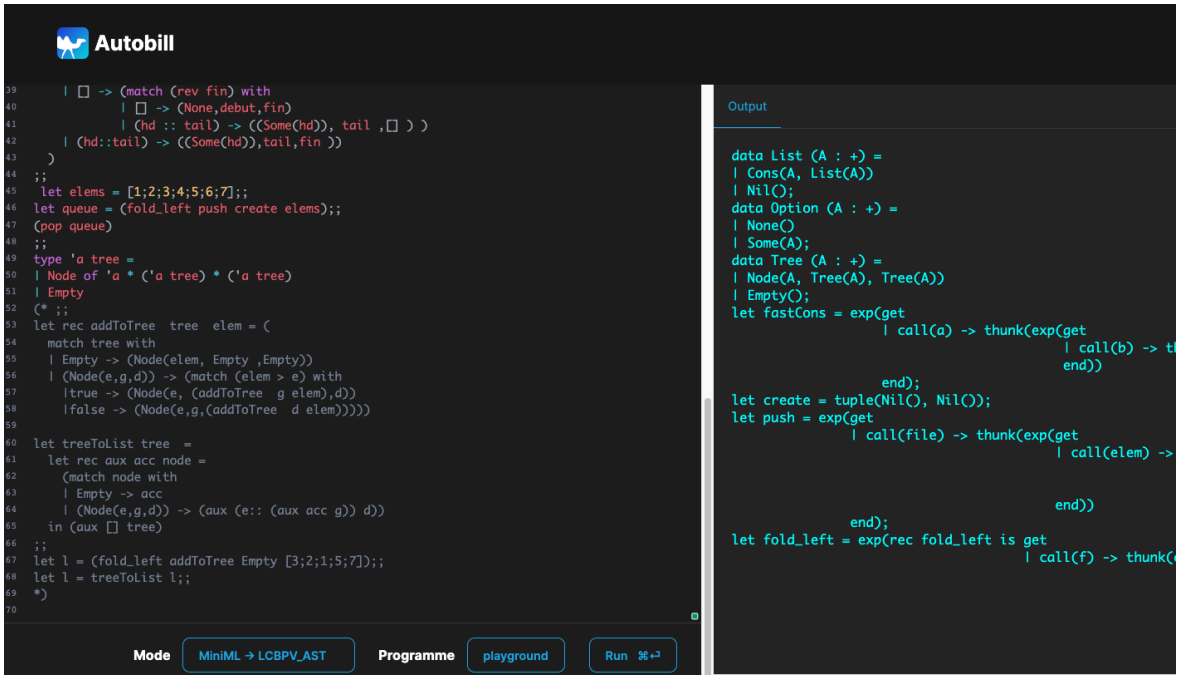
2.1 Client uniquement

2.1.1 Outils et Technologies utilisés

- **HTML / CSS / Javascript (JS)** : 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.js est une bibliothèque JavaScript open-source pour la création d'interfaces utilisateur,utilisée pour la création d'applications web modernes et interactives. Parmi les avantages de cette technologie, il y a l'utilisation du Virtual DOM (Document Object Model) qui permet une mise à jour plus efficace et rapide des éléments d'une page. Le Virtual DOM est une représentation virtuelle d'un arbre DOM qui est stockée en mémoire et mise à jour en temps réel en fonction des interactions de l'utilisateur avec l'interface. On modifie seulement les éléments impactés, et non l'ensemble du DOM de la page, ce qui se traduit par des temps de réponse plus rapides et des meilleures performances. Aussi, React est basé sur la programmation orientée composant. L'interface utilisateur est décomposée en petits composants réutilisables, chacun étant responsable de l'affichage d'une partie spécifique de l'interface. Chaque composant est construit de manière indépendante et peut être utilisé à plusieurs endroits dans une application. Cette approche modulaire rend l'interface plus flexible et maintenable.
- **CodeMirror** : C'est une librairie Javascript permettant d'intégrer un éditeur de code puissant, incluant le support de la coloration syntaxique, de l'autocomplétion ou encore le surlignage d'erreurs. Les fonctionnalités de l'éditeur sont grandement extensives et permettant même la compatibilité avec un langage de programmation personnalisé comme **MiniML**. Enfin, CodeMirror est disponible sous licence MIT, libre de droits.

- **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. L'API de Js_of_Ocaml est suffisamment fournie pour développer entièrement des applications web complètes et fonctionnelles. Pour ce projet, il sert surtout pour interagir avec Autobill et la librairie de MiniML depuis le client Web. Dans un fichier `main.ml`, on exporte un objet Javascript contenant plusieurs méthodes correspondant chacune à un mode d'exécution différent d'Autobill. Chaque méthode prend en entrée le code MiniML à traiter et réalise les transformations nécessaires pour générer la sortie demandée. Néanmoins, en l'absence de sortie standard ou d'erreurs, les messages d'exceptions d'Ocaml, par exemple, n'apparaissent que dans la console Javascript du navigateur. Js_of_ocaml met à notre disposition un module `Sys_js` qui offre des primitives permettant de capturer les possibles messages sur les sorties et les rediriger dans des buffers. Ces buffers peuvent être convertis en chaînes de caractères et retournés au client par la suite.
- **MiniZinc**: À la génération des expressions de contraintes, Autobill retourne une sortie au format MiniZinc. Ce langage permet de décrire des problèmes de manière déclarative à l'aide de contraintes logiques. L'objectif avec MiniZinc est de calculer les bornes mémoires minimums pour satisfaire les contraintes mémoires du programme et d'afficher, sous forme d'équation, le résultat dans la sortie de notre IDE. Son API prend en charge une large gamme de solveurs. Aussi, il dispose d'une grande communauté d'utilisateurs et de contributeurs, ce qui nous permet de trouver nombreuses ressources disponibles pour l'apprentissage et le dépannage. Sa librairie est codée en C++ mais il reste utilisable dans notre interface Web grâce à Web Assembly. C'est un format binaire de code exécutable qui permet de porter des applications codées dans des langages de programmation sur le Web. Grâce à des compilateurs vers Web Assembly, comme Emscripten pour C/C++, on peut lancer des tâches intensives de résolution de contraintes, avec des performances proches du natif, depuis n'importe quel navigateur Web moderne.

2.1.2 Aperçu de l'interface graphique

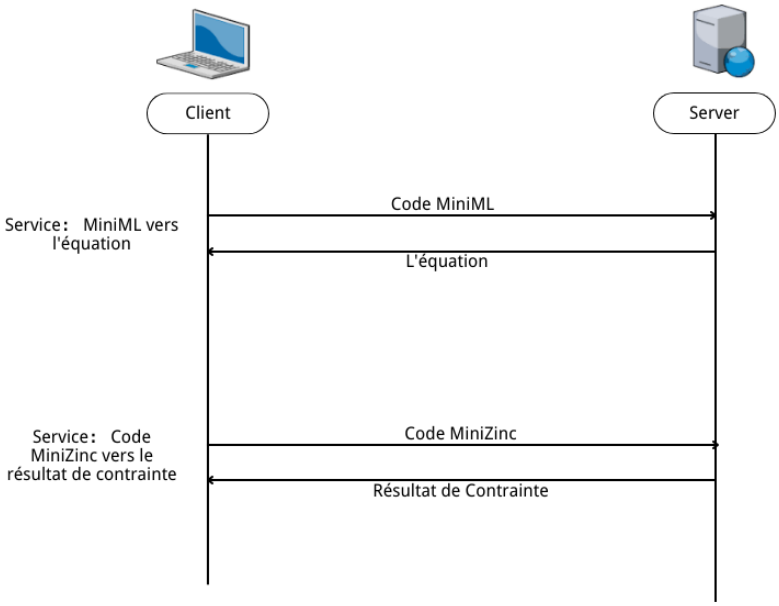


2.1.3 Tâches réalisées

- Intégration d'un 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 de traitement du code **MiniML** :
  - Affichage de l'AST MiniML
  - Affichage de l'AST de **Call-By-Push-Value**
  - Affichage de l'équation résultant de l'analyse statique
  - Vers Représentation Interne **Autobill**
- Remontée d'erreurs et affichage dynamique sur l'interface
- Implémentation du solveur d'équations MiniZinc côté client

2.2 Serveur + client

2.2.1 Schéma de communication



2.2.2 Outils et technologies utilisés

2.2.2.1 Coté client

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

2.2.2.2 Coté serveur

- **NodeJS**: NodeJS permet une gestion asynchrone des opérations entrantes, ce qui permet d'exécuter plusieurs opérations simultanément sans bloquer le fil d'exécution principal. Par exemple, si deux requête sont envoyé au serveur en même temps, elles seront gérées en parallèle par le serveur. Ainsi, grâce à cette gestion asynchrone, NodeJS permet d'optimiser l'utilisation des ressources système en réduisant les temps d'attente et en évitant les blocages inutiles, ce qui peut augmenter l'efficacité et les performances du programme. 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. Par example, le

module "Child Processes" nous permet d'exécuter le code MiniZinc en passant les commandes directement. Cela nous permet d'éviter les restrictions du côté full-client au niveau du résolveur de contraintes.

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 de traitement du code **MiniML** :
  - Affichage de l'équation résultant de l'anlyse statique
- Implémentation du solveur d'équations MiniZinc côté client et server

3 MiniML

3.1 Pourquoi MiniML ?

MiniML émerge 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 afin de pouvoir manipuler son entrée en **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*. La différenciation entre ces 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**. De plus tout code **MiniML** est parfaitement compatible avec un parseur ou compilateur **OCaml**.

3.2.1 Contenu actuel

- Integer
- Boolean
- Listes
- Fonction Récusives
- Opérateurs de Bases
- Construction de Types de Données

- Types de Données Paramétrés
- Types de Données Récursifs
- Variables Globales/Locales
- Files *(FIFO)*

3.2.2 Dépendances

- **Menhir** : *Menhir* est l'unique dépendance de l'implémentation de **MiniML**, Cette librairie permet la génération d'analyseurs syntaxiques en OCaml nous permettant d'éviter le développement d'un analyseur syntaxique rigide. Cette décision compatible avec les deux architectures du projet, nous a permis d'économiser en temps de développement et gagné en flexibilité.
  Menhir est disponible sous licence GPL

3.3 Un exemple de code MiniML

Cet exemple de code **MiniML** est une implémentation possible d'une file d'attente sans effet de bord en **MiniML**.

```
type 'a option =
| None
| Some of 'a
;;

let createQueue = ([],[]);;

let push file elem =
(match file with
| (a,b) -> (a,(elem::b)))
;;

let pop file =
  (match file with
  | (debut, fin) ->
    (match debut with
    | [] -> (
              match (rev fin) with
              | [] -> (None,debut,fin)
              | (hd :: tail) -> ((Some(hd)), tail ,[])
            )
    | (hd::tail) -> ((Some(hd)),tail,fin ))
  )
;;
```

```
let elems = [1;2;3;4;5;6;7];;
let queue = (fold_left push createFile elems);;
(pop queue)
```

3.3.1 Schema de traduction

Dans le prochain rapport, nous allons nous baser sur une variante de cet exemple pour décrire, avec des schémas de traduction comment l'on passe de **MiniML** à **Call-By-Push-Value** compatible pour **Autobill**.

4 Conclusion et tâches à réaliser

4.1 Conclusion

La réalisation de cette interface a fait intervenir un large panel de sujets en lien avec la formation du Master d'informatique STL et mis à profit les connaissances acquises lors de ce semestre. Le projet est à un stade d'avancement satisfaisant. Autobill étant encore en phase expérimentale, celui-ci ajoute contiuellement des nouveautés et corrections que l'on doit intégrer.

La suite consistera surtout à consolider les bases établies sur tous les aspects du projet présentés dans ce rapport et les adapter aux changements d'Autobill. Aussi, il serait intéressant à titre de démonstration de comparer notre solution avec celle de Jan Hoffmann et l'interface de RAML [3], mentionnée en section 1.

4.2 MiniML

- Ajout de sucre syntaxique. (Records, Operateurs Infixes, ...)
- Ajout d'une librairie standard.
- Spécification complète du langage.
- Bibliothèque de structures de données complexes
- Régles de traduction de **MiniML** vers **Autobill**
- Schemas de traduction d'une structure *FIFO* vers **Autobill**

4.3 Serveur

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

4.4 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.

4.5 Tests

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