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

Dans le cadre de sa thèse sur l'analyse statique de la consommation mémoire d'un programme au sein de l'équipe APR du LIP6, notre tuteur de projet, Hector Suzanne, a développé [**Autobill** @autobill].

L'analyse statique est un domaine de l'informatique qui consiste à mesurer 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 pour répondre à une problématique. Dans le cas d'Autobill, nous nous sommes particulièrement intéressés à l'occupation 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 @Hoffmann] (*Automatic Amortized Resource Analysis*). Des solutions se basant sur ces théories existent, comme [Resource Aware ML @RAML], 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, parmi les entrées possibles illustrées sur la gauche de la figure 1, Autobill ne supporte uniquement que des programmes écrits soit en modèle machine propre à Autobill, soit en [**Call-By-Push-Value** (CBPV) @Levy], avec ou sans continuation explicite.

Call-By-Push-Value est un langage qui utilise un paradigme déjà éprouvé, décrit dans la thèse de [Paul Blain Lévy @Levy]. 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. Ces atouts font de CBPV un langage de choix à analyser pour Autobill.

L'entrée est donc imposée. Ainsi, pour étendre l'usage d'Autobill à un autre langage de programmation, un travail de traduction de ce langage donné vers CBPV doit avoir lieu. Cela implique donc de comprendre le langage que l'on compile, notamment les stratégies d'évaluations implicites mises en œuvre, et de l'adapter aux caractéristiques uniques de CBPV citées plus haut.

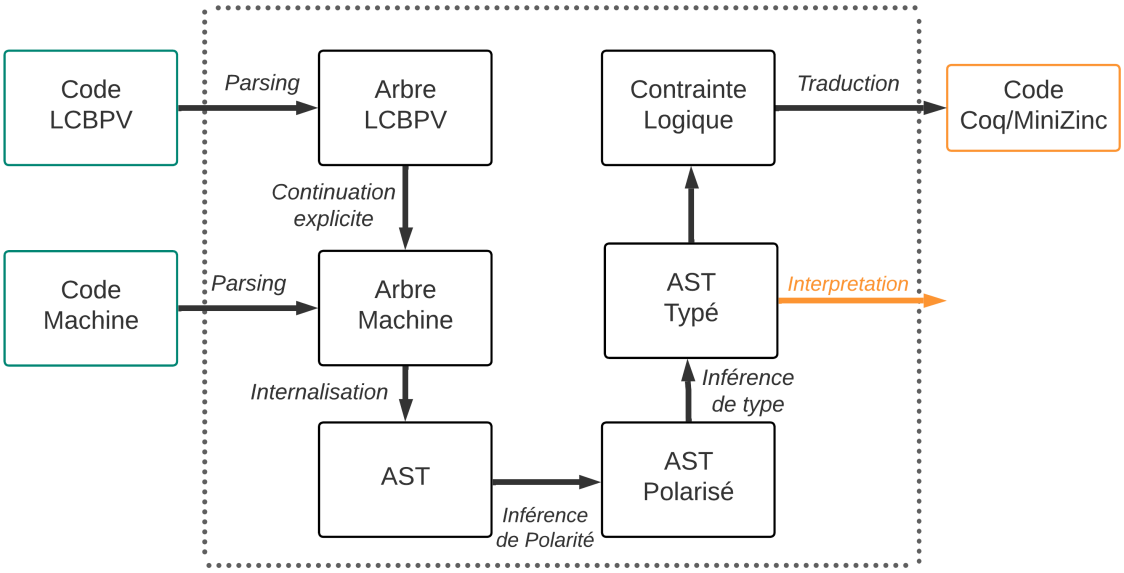

Figure 1: Représentation simplifiée d'Autobill

À partir d'une entrée en CBPV, Autobill traduit le programme en un code machine avec continuations explicites, 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, pour démarquer les calculs et les valeurs dans l'AST. Enfin, en sortie, on remarque dans la figure 1 la possibilité de tirer une interprétation du programme, mais surtout de récupérer les contraintes dans un format [MiniZinc @minizinc] ou [Coq @coq]. Ce sont des outils des assistants de preuve qui permettent, à l'aide d'un langage dédié, d'exprimer explicitement des contraintes logiques. Autobill s'en sert pour décrire les bornes mémoires nécessaires au fonctionnement d'un programme. On peut alors traiter ces équations avec des solveurs, fournis aussi par ces deux outils, pour prouver des propriétés de complexité temporelle ou spatiale.

1.3 Objectifs du projet

Notre démarche se rapproche de celle de [RAML @RAML] avec leur site officiel: offrir une interface Homme-Machine accessible à tous et illustrant un sujet de recherche en analyse statique.

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.

Par rapport à la chaîne d'instructions d'Autobill et à la Figure 1, on se place donc en amont du code LCBPV en entrée et après la sortie en code MiniZinc/Coq.

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

- L'implémentation du langage MiniML et sa traduction vers CBPV
- 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

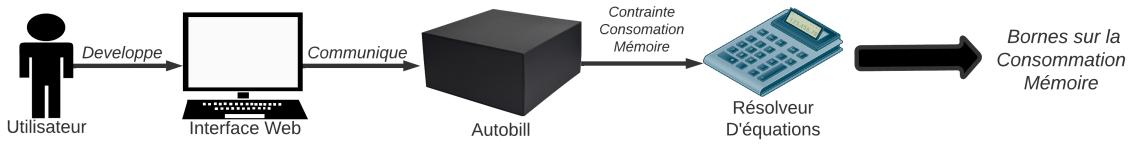

Figure 2: Représentation du système cible

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.

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 solveur 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.

De cette démarche, il en résulte un code source d'environ 500 lignes, client et serveur compris, faisant tourner notre IDE en ligne dans un état fonctionnel.

2.1 Client uniquement

Une première approche tout client a été mise œuvre dès le début du projet. Celle-ci permettait de garantir une facilité dans le déploiement en ligne de notre solution. L'environnement limité du navigateur Web a remis en question la tenue de cette architecture, mais des compromis ont été trouvés pour satisfaire la contrainte d'interopérabilité entre le Web et Ocaml.

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]: 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.

D'autres solutions concurrentes et valables telles que [Angular.js @angularjs] ou [Vue.js @vuejs] proposent une expérience développeur similaire.
Néanmoins, ces dernières sont bien moins recherchées sur le marché du travail que React. Nous voulions à travers ce projet nous former et pratiquer une technologie très en vogue et que l'on pourra facilement réinvestir plus tard dans notre parcours professionnel.

- [**CodeMirror** @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** @Minsky_Ocaml @chailloux @Leroy + [**Js_of_OCaml** @js_of_ocaml]: Afin de manipuler la librairie d'Autobill, il est nécessaire de passer par 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.
  La question s'est posée de l'intérêt de Js_of_OCaml comparé à d'autres technologies comme [ReasonML @reasonml] ou [Rescript @rescript]. En effet, ce sont tout deux des langages qui ont émergé d'Ocaml et permettent de créer dans un paradigme fonctionnel des applications web complexes. Des compilateurs pour transpiler du code Rescript (Bucklescript) ou ReasonML vers Javascript existent et ReasonML permet même de compiler vers du code en React.
  Néanmoins, notre objectif principal est la manipulation de la librairie d'Autobill ainsi que celle de MiniML depuis le Web. Ces deux technologies affichent des syntaxes différentes de celle d'OCaml, ce qui empêche toute compatibilité avec les bibliothèques d'OCaml. Js_of_OCaml en complément avec un client en React

correspond donc à un bon compromis dans notre cas d'étude.

- **MiniZinc** @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, depuis n'importe quel navigateur Web moderne.

2.1.2 Aperçu de l'interface graphique

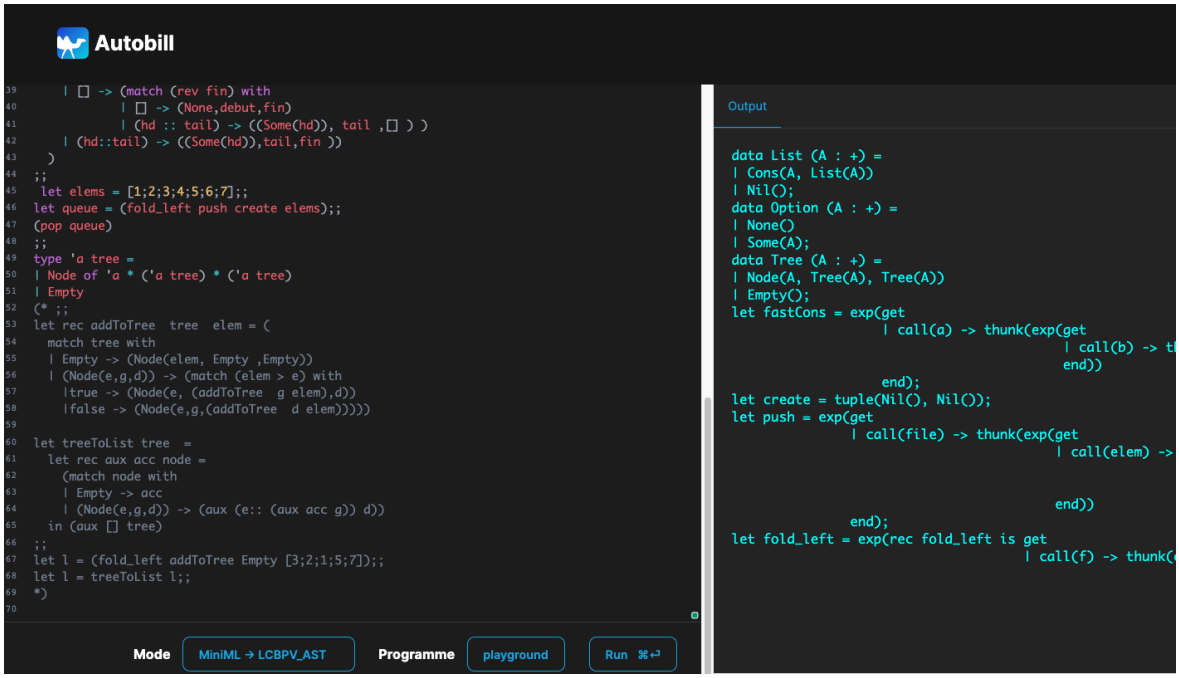

Figure 3: Aperçu de l'interface graphique

2.2 Serveur + client

Dans le stade actuel d'Autobill, une architecture avec un client seul peut répondre aux exigences du projet STL. Néanmoins, Autobill évolue constamment et rien ne garantit que ses itérations suivantes puissent être supportées par notre solution. Dans l'optique de rendre notre solution plus flexible et *futureproof*, une nouvelle version de notre interface, qui déporte les tâches complexes vers un serveur distant, a été développée.

On a souhaité aussi adapter le client pour qu'il opère dans ces deux architectures différentes. Ainsi, dans notre environnement de développement, on peut facilement faire la bascule entre un mode de fonctionnement local/synchrone et un mode distant/asynchrone.

2.2.1 Schéma de communication

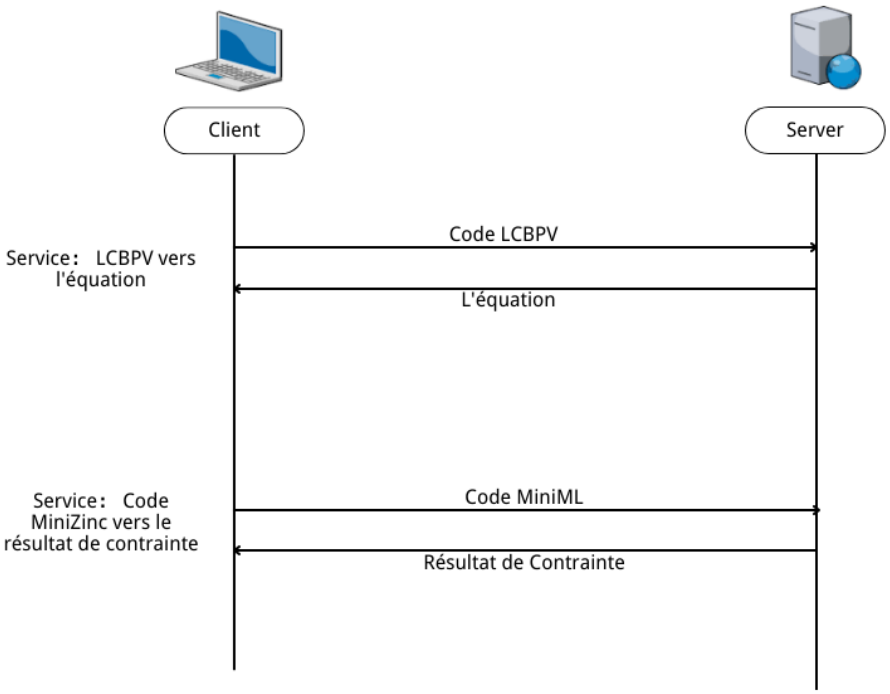

Figure 4: Schéma de communication

Dans notre architecture Client-Server, nous avons mis en place 2 services principaux en utilisant le protocole HTTP (voir Figure 4): la transformation de code MiniML vers l'équation résultant de l'analyse statique et l'exécution de code MiniZinc. Dans le premier service, le client envoie le code LCBPV au serveur via une requête POST, le serveur le convertit en équation et le renvoie au client. Dans le second service, le client envoie le code de MiniZinc au serveur via une requête POST. Le serveur passe le code à un solveur de MiniZinc et renvoie le résultat au client.

2.2.2 Outils et technologies utilisés

- **Node.js** @Node_js : Node.js 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 requêtes sont envoyées au serveur en même temps, elles seront gérées en parallèle par le serveur. Ainsi, grâce à cette gestion asynchrone, Node.js 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.
  De plus, 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 @Child_Processes] nous permet d'exécuter le code MiniZinc en passant les commandes directement. Cela nous permet d'éviter les restrictions du côté tout-client au niveau du solveur de contraintes notamment. Enfin, un des avantages de Node.js est qu'il nous permet d'utiliser le même langage de programmation que le client. On s'évite ainsi les écueils autour de l'interopérabilité et de la compatibilité entre deux instances codées dans des langages différents.

- **Express.js** @Express_js : Express.js est une bibliothèque d'application web populaire basée sur la plateforme Node.js, utilisé pour construire des applications web et des API évolutives. Il fournit de nombreux middlewares, tels que [morgan @morgan] pour enregistrer les journaux de session HTTP et [helmet @helmet] pour garantir la sécurité. Avec Express.js, nous pouvons facilement ajouter des middlewares en utilisant directement `app.use()` sans avoir à les ajouter manuellement. De plus, Express.js dispose d'un puissant routeur qui permet aux développeurs de gérer facilement les routes et de construire des API REST de manière efficace. Par conséquent, l'utilisation d'Express.js rend la développement plus facile et plus efficace, et rend également le code plus concis. Donc il permet aux développeurs de créer facilement des applications web plus rapides et évolutives. Il dispose d'une plus grande communauté que les bibliothèques plus jeunes, avec de nombreuses ressources et solutions disponibles.

- **REST API** @PC3R : L'API REST est un modèle de conception d'interface de programmation d'application Web (API) utilisé pour fournir un accès aux ressources sur le Web. Il est basé sur le protocole HTTP et utilise des requêtes et des réponses HTTP pour communiquer.
  La conception de l'API REST est très simple. Elle utilise des verbes HTTP (GET, POST, PUT, DELETE, etc.) pour représenter les opérations effectuées, ainsi que des formats de données standards (tels que JSON, XML) pour la transmission de données. L'état des réponses est représenté sous la forme de codes d'état HTTP. Par exemple, le code 200 représente le succès, le code 404 représente la ressource introuvable, etc.
  Cela permet aux clients de rapidement déterminer le résultat de la réponse en fonction du code d'état, sans avoir besoin de parser des informations de réponse complexes. Par rapport à "SOAP" qui ne peut utiliser que XML pour transférer des informations, l'API REST simplifie et facilite l'échange de données entre les clients et les serveurs.
- **Morgan** @morgan : Morgan est un middleware JavaScript permettant d'enregistrer les journaux de session HTTP. Elle peut produire des informations dans plusieurs endroits (la console, les fichiers, les en-têtes HTTP...) pour nous aider à déboguer et à analyser le fonctionnement de leurs applications. De plus, Morgan peut être facilement ajouté aux applications Node.js en installant simplement les dépendances à l'aide de npm et en les introduisant dans l'application à l'aide d'[Express.js @Express_js]. Par rapport à d'autres outils similaires, Morgan est plus facile à utiliser, car il ne nécessite pas d'écriture de code supplémentaire. Ainsi, Morgan produit une plus grande variété des cibles de sortie.
- **Helmet** @helmet : Helmet est un middleware pour les applications Node.js, conçu pour aider à protéger les applications Web contre certaines attaques Web courantes. De plus, l'utilisation de Helmet est très facile, en installant simplement les dépendances à l'aide de npm et en les introduisant dans l'application à l'aide d'[Express.js @Express_js]. Et Helmet est un middleware populaire pour Node.js, bénéficiant d'une maintenance actifs de la communauté, ce qui nous permet de bénéficier facilement d'aide et de support dans la communauté.

2.3 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
- Mise en place d'un serveur distant manipulant les libraires OCaml du projet et le solveur MiniZinc
- Exposition du serveur avec une API REST
- Mise en relation du client et du serveur

3 MiniML

3.1 Description de MiniML

MiniML émerge du choix par nos encadrants de créer un langage fonctionnel simple et accessible pour les utilisateurs d'Autobill servant d'abstraction à **CBPV**. Dans le cadre de ce projet MiniML, dispose d'une implémentation écrite en **OCaml**. Nous avons pris la décision de rendre la syntaxe MiniML parfaitement compatible avec OCaml simplifiant les comparaisons avec [RAML @RAML].

Le développement de **MiniML** suivant les besoins de nos encadrants celui-ci est pour l'instant sans effets de bord.

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*. Dans CBPV, une distinction a lieu entre les calculs et les valeurs permettant de décider en détail comment ceux-ci sont évalués. Nous permettant, lors de la traduction depuis un autre langage, de choisir le type de stratégie utilisée.

3.1.2 Contenu actuel

Le contenu actuel de **MiniML** est divisé en deux. Une partie noyau qui contient les éléments de base du langage servant de briques de construction pour la second partie ou se trouvent les types de données et les fonctions de haut niveau.

Le noyau de **MiniML** contenant deux types de base les entiers et les booléens ainsi que les opérateurs de base. À partir de ces éléments, il est possible de construire des types de données plus complexes en exprimant des types paramétrés et en les combinant. Les listes sont un exemple de typique de structures de données construit à partir de ces mécanismes. Comme dans tout langage fonctionnel, il est possible de définir des fonctions anonymes ou non et de les passer en paramètre à d'autres fonctions. Il est également possible de définir des variables globales et locales.

La partie contenant les types de données et les fonctions de haut niveau est la plus importante, car elle agit en tant que vitrine d'**Autobill** avec pour but de fournir un ensemble de types de données complexes dont l'analyse amortie est possible et mettant en lumière les apports d'**Autobill**.
Toujours en cours de développement, cette partie, basée sur le noyau, contient pour l'instant trois types de données les listes, les files et les arbres binaires.

3.1.3 Dépendances

À l'instar du développement de l'interface web, la question des dépendances est cruciale pour MiniML. En effet pour n'avoir qu'une seule implémentation de MiniML et donc limiter l'effort de développement, il est nécessaire de ne choisir que des librairies qui sont parfaitement compatibles avec les deux architectures du projet.

- [**Menhir** @menhir]: *Menhir* est l'unique dépendance de l'implémentation de MiniML. C'est une librairie qui génère des analyseurs syntaxiques en OCaml et nous évitant le développement d'un analyseur syntaxique rigide. C'est à la suite de différents tests de compatibilité avec les deux architectures du projet que nous avons choisi cette librairie nous permettant un gain en temps et en flexibilité non négligeable. Menhir est disponible sous licence GPL

3.2 Un exemple de code MiniML

Cet exemple est une implémentation possible d'une file d'attente en **MiniML**. Dans le prochain rapport, nous allons nous baser sur une variante de ce code pour décrire, avec des schémas de traduction basés sur la spécification du langage, comment l'on passe d'un code **MiniML** à un code **Call-By-Push-Value** reçu en entrée par **Autobill**.

Le choix de ce code est motivé par le fait qu'il est assez simple et que ce dernier peut mettre en avant les résultats plus précis qu'une analyse amortie permet d'obtenir.

```
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)
```

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. En premier lieu, le cours d'analyse de programme statique @APS pour toute la partie MiniML et le processus de transformation vers CBPV. Puis, les cours de programmation concurrente répartie, réactive et réticulaire @PC3R, notamment pour la partie réticulaire et l'architecture d'applications Web modernes.

Le projet est à un stade d'avancement satisfaisant. Autobill étant encore en phase expérimentale, celui-ci est alimenté continuellement de 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 @RAML], 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
- Règles de traduction de **MiniML** vers **Autobill**
- Schémas de traduction d'une structure *FIFO* vers **LCBPV**

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**