Sorbonne Université

Rapport Final

19 Mai 2023

Projet STL :
-
Interface pour Autobill

Auteurs :
Brahima Dibassi
Zeid Fazazi
Yukai Luo

Encadrants :
Hector Suzanne
Emmanuel Chailloux

Table des matières

1 Contexte du projet

Dans le cadre de sa thèse sur l'analyse statique de consommation mémoire d'un programme, notre tuteur de projet, Hector Suzanne, membre de l'équipe APR du laboratoire LIP6, a développé **Autobill**.
L'analyse statique est un ensemble de méthodes formelles permettant de mesurer et détecter automatiquement des comportements ou des erreurs dans un programme en examinant son code source. Parmi les usages les plus courants de l'analyse statique, il y a le débogage pour identifier des erreurs syntaxiques (fautes de frappe) ou l'usage de variables non déclarées ou non-initialisées. D'autres outils emploient des heuristiques et des règles pour répondre à des problématiques d'optimisation de code ou de fiabilité/sécurité. Ces analyseurs statiques sont conçus dans le but d'améliorer la maintenabilité du code.
Autobill s'intéresse particulièrement 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 et Steffen Jost sur l'analyse de consommation de ressources automatisé AARA [2] (*Automatic Amortized Resource Analysis*).
Cette technique suit un ensemble de règles d'inférences : chaque trait du langage analysé est annoté par ses bornes mémoire et on peut déduire la consommation mémoire que ce trait va engendrer. Pour déterminer ces bornes, l'AARA s'aide de l'analyse amortie par la méthode du potentiel. En effet, l'analyse amortie exploite la structure séquentielle des opérations sur des structures de données dynamiques pour calculer le coût total de cette séquence. Avec la méthode du potentiel, on attribue un potentiel aux structures de données manipulées, ce qui va affecter le coût que va avoir une opération sur cette structure. L'application de ces règles d'inférence sur l'ensemble des expressions du programme fournit une estimation correcte de ses bornes mémoires.

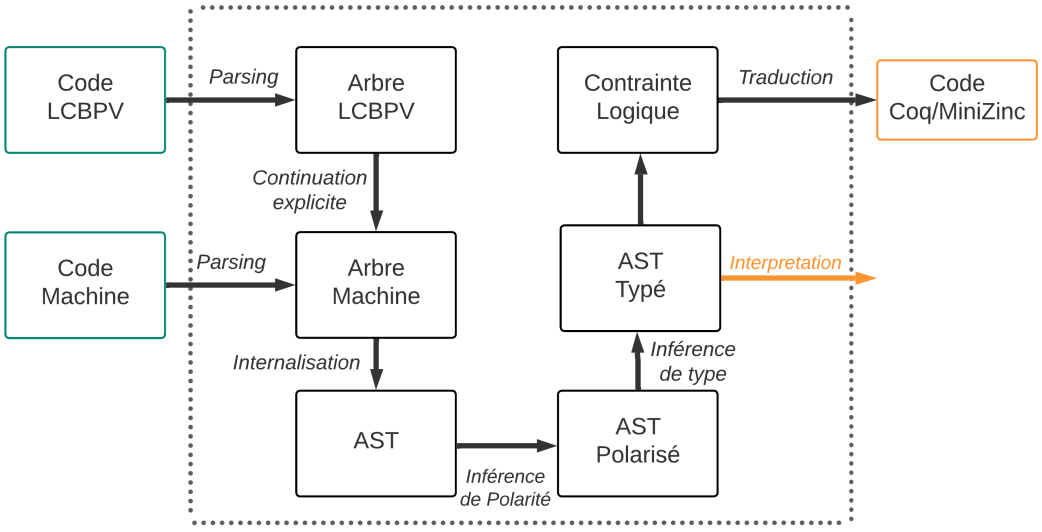

FIGURE 1 – Représentation simplifiée d'Autobill

1.1 Historique et définitions

Des langages de programmation expérimentaux ont vu le jour et ont implémenté cette analyse comme Resource Aware ML [3], un langage fonctionnel à la ML créé par Jan Hoffmann, Klaus Aehlig et Martin Hofmann.

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 par rapport à RAML. D'abord, parmi les entrées possibles illustrées sur la gauche de la figure **??**, Autobill ne supporte uniquement que des programmes écrits soit en modèle machine propre à Autobill, soit en **Call-By-Push-Value** (CBPV) [4], 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 [4]. Dans CBPV, toutes les valeurs et fonctions sont stockées dans une pile. Lorsqu'une fonction est appelée, ses arguments sont placés sur la pile avant que la fonction elle-même ne soit ajoutée à la pile. Ce mécanisme permet de 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 : soit en ***call-by-value***, en évaluant les arguments avant de lancer l'opération, soit en ***call-by-name***, en évaluant les arguments uniquement lorsqu'ils seront effectivement utilisés dans la fonction appelée. Ainsi, on fixe quand les évaluations se déroulent, afin de mieux prédire la consommation de mémoire à chaque étape du programme. Ces traits 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.

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 **??** la possibilité de tirer une interprétation du programme, mais surtout de récupérer les contraintes dans un format MiniZinc [5] ou Coq [6]. 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.

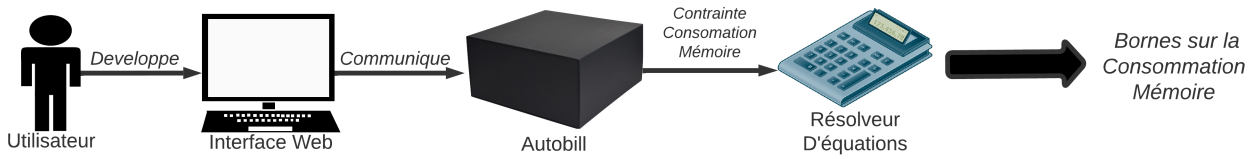

FIGURE 2 – Représentation du système cible

1.3 Objectifs du projet

Notre démarche se rapproche de celle de RAML [3] 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 **??**, 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

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

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. Cette partie se concentra sur la présentation de notre client Web, de l'implémentation des fonctionnalités importantes et des problèmes rencontrés ainsi que leurs résolutions.

2.1.1 Interopérabilité Web-Ocaml

Autobill est un projet entièrement codé en Ocaml, un langage de programmation multiparadigme compilé, et notre projet STL impose un environnement sur navigateur Web, fonctionnant exclusivement avec son langage de script Javascript. Avant de commencer tout codage, il est nécessaire de passer en revue l'état de l'art autour de la compilation et d'exécution de programmes Ocaml sur le Web. De ces recherches vont découler des choix de conception qui vont nous impacter tout le long du semestre.

D'abord, il y'avait la piste des compilateurs vers WebAssembly. WebAssembly est un standard du W3C (World Wide Consortium) qui regroupe un bytecode (.wasm) et un environnement d'exécution compatible avec les navigateurs modernes Javascript. Le bytecode étant une représentation très bas niveau de l'exécutable, l'idée principale est de compiler un langage de programmation plus haut niveau, comme C/C++, Rust,.. vers bytecode. Ce bytecode est par la suite compilé par l'environnement d'exécution dans le langage machine de l'hôte. Tant que le support d'un langage de programmation est garanti par un compilateur WebAssembly, il est possible de programmer des applications web complètes dans le langage de son choix ou réinvestir des programmes codés dans ces langages dans une application codée avec des technologies Web.
Pour le cas d'Ocaml, cette possibilité est offerte grâce au post-processeur Wasicaml : il offre un binaire qui prend en entrée un bytecode Ocaml et un fichier .wasm de destination pour traduire chaque instruction dans son équivalent en WebAssembly. Ainsi, on pourrait générer nos fichiers .wasm à partir des binaires d'Autobill et de notre compilateur MiniML, les intégrer à notre applicatiton Web et les appeler avec Javascript en leur fournissant les options nécessaires. Cette solution demande néanmoins une dépendance à un projet encore en phase expérimentale, avec certains traits d'Ocaml non intégrés (notamment certaines fonctions du module Unix).

```
$ ocamlc -o hello hello.ml
$ wasicaml -o hello_wasm hello
```

FIGURE 3 – Chaine d'instructions pour générer le bytecode WebAssembly

La seconde piste émise par nos tuteurs de projet était l'utilisation de compilateurs Ocaml vers Javascript. La présentation du projet nous a notamment pointé vers Js_of_Ocaml. C'est une librairie contenant, entre autres, un compilateur qui transpile du bytecode OCaml en Javascript et propose une grande variété de primitives 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. On profite alors de l'expérience développeur offerte par Ocaml avec son style de programmation fonctionnelle en y intégrant les traits nécessaires pour construire des pages Web dynamiques.

Il est aussi possible de l'utiliser en coopération avec une base de code Javascript déjà rédigée grâce à la fonctionnalité d'export offerte par Js_of_Ocaml. En effet, on pourrait avoir dans un objet Javascript 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. On peut aussi tirer profit des exceptions d'Ocaml et de la capture des sorties d'erreurs offerte par Js_of_Ocaml pour rediriger les messages d'erreurs et les afficher à l'utilisateur. Une fois l'objet exporté et le fichier Ocaml compilé vers Javascript, n'importe quel fichier Javascript du projet pourra ensuite importer l'objet et accéder aux méthodes définies précédemment. On garde ainsi une liberté sur le choix de technologies pour construire notre application.

```
Js.export
    "ml"
    (object%js
       method translate code =
         let stderr_buff = Buffer.create 100 in
         Sys_js.set_channel_flusher stderr (Buffer.add_string
            stderr_buff);
         let lexbuf = Lexing.from_string ~with_positions:true (Js.
            to_string code) in
         let res = string_of_lcbpv_cst (translate_ML_to_LCBPV
            lexbuf)
         object%js
           val resultat = Js.string res
           val erreur = Js.string (Buffer.contents stderr_buff)
         end
    end)
```

FIGURE 4 – Création et exportation d'un objet Javascript avec une méthode de traduction de MiniML vers CBPV

Enfin, une dernière piste suggérée a été l'utilisation de langages de programmation camélien compilables vers Javascript, comme ReasonML ou 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. Ils profitent d'une syntaxe ML, d'outils et d'un support communautaire intéressant, faisant des deux langages des alternatives intéressantes pour le Web. Néanmoins, notre objectif principal est la manipulation de la librairie d'Autobill ainsi que celle de MiniML depuis le Web. La compatibilité avec les librairies en Ocaml n'est cependant pas garantie dans le contexte d'une compilation vers Javascript.

Notre groupe a donc opté pour un choix de conception assez flexible afin de nous adapter aux situations que l'on pourrait rencontrée dans le projet. Le client fonctionnera conjointement grâce à une partie Ocaml servie à l'aide de Js_of_Ocaml et une partie en Javascript, plus précisément en React.js, pour assurer les fonctionnalités propres à notre IDE.

2.1.2 L'éditeur de code

L'idée initiale était de proposer une interface de développement intégrée sur le Web. Celle-ci devait proposer les outils nécessaires pour réaliser les trois tâches suivantes : analyser, écrire et déboguer du code. Pour répondre à ces besoins, plusieurs composants essentiels vont devoir fonctionner ensemble pour fournir l'expérience d'un IDE classique :

- Un éditeur de code pratique et adapté à l'écriture de code dans notre langage MiniML.
- Un menu de navigation entre les programmes proposés par l'application ou créés par l'utilisateur
- Un menu de sélection entre différentes configurations d'analyses
- Une sortie standard et d'erreur pour afficher le retour de l'analyse d'Autobill

La question de l'éditeur de code s'est rapidement posée pour nous. En effet, c'est sur ce composant que va être concentrée l'expérience de l'application et nous avions la volonté de nous rapprocher au plus d'une prise en main similaire aux IDE modernes (Visual Studio Code, Eclipse, IntelliJ...). L'idée d'utiliser un simple champ de texte agrandi n'était donc pas envisageable.

En premier lieu, l'effort de développement a été mis dans la construction *from Scratch* d'un éditeur de code. Cela a permis de fixer clairement les exigences et les fonctionnalités que nous attendions d'un éditeur de code moderne. Déjà, il devait proposer une expérience d'écriture correcte avec toutes les manipulations usuelles et attendues : le comptage des lignes, la recherche et le remplacement rapide d'une suite de caractères (Ctrl+F), le *linting* d'erreurs... Enfin, il y'a le support du langage d'écriture des programmes. Par cela, il est question notamment de coloration syntaxique pour isoler les mots clés et symboles réservés du langage mais aussi de l'autocomplétion basique.

L'examen de nos besoins nous a donc contraints à repousser l'idée d'une création d'un éditeur en partant de zéro. Délivrer les fonctionnalités citées plus haut tout en assurant la stabilité de la solution serait complexe et concentrerait beaucoup d'efforts sur ce composant, au détriment de chantiers clés comme le MiniML. C'est un élément de l'application d'autant plus important qu'il ne doit pas être négligé et demande à lui seul l'effort et le temps nécessaires à un projet universitaire. pour être réalisé. Nous nous sommes donc tournés vers des librairies et solutions déjà existantes

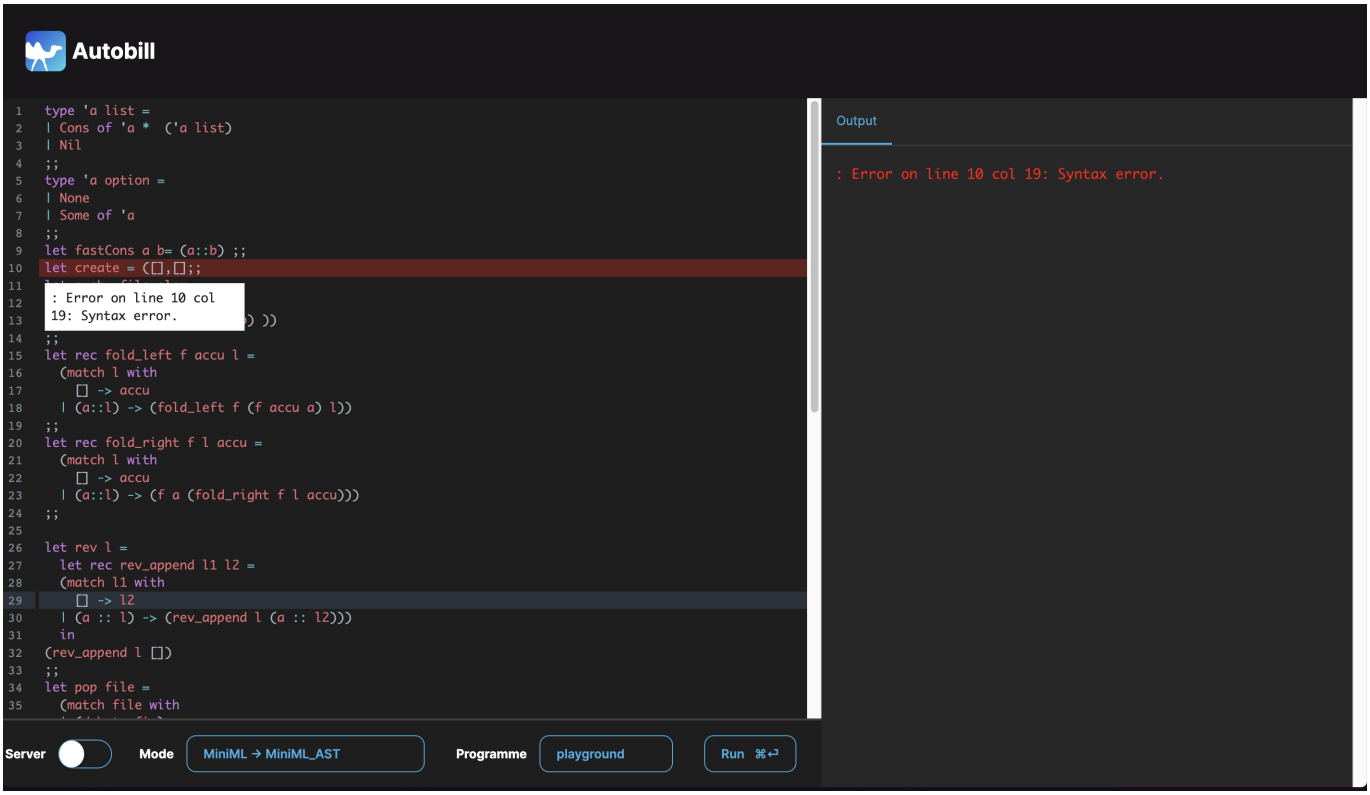

FIGURE 5 – Interface pour Autobill, dans le scénario d'une erreur

afin de les intégrer dans notre projet.

Parmi les librairies disponibles, nous avons choisi CodeMirror. C'est un éditeur de code pour navigateur Web, disponible en libre de droits, qui peut s'intégrer à n'importe quelle application Web. Ses fonctionnalités sont exhaustives et il offre de multiples choix de personnalisations pour l'adapter à un langage spécifique. Nativement, il y'a le support intégré pour la syntaxe de langage à la ML comme Ocaml ou F# : MiniML étant un substrat, on peut aisément créer notre fichier de configuration pour ne garder que les mots clés essentiels de notre langage et Codemirror s'occupe du parsing du code et de la coloration de chaque caractère.

Enfin, le *linting* est disponible et permet de dynamiquement surligner des lignes de l'éditeur. Comme illustré dans la Figure 3, on peut, à la volée, afficher directement les parties du programme concernées par une erreur de l'analyse. En effet, quand une méthode de Js_of_Ocaml est appelée et qu'une exception est levée durant l'exécution, sa sortie d'erreur peut être redirigée et retournée directement au client. Côté Ocaml, on peut alors formater la sortie d'erreur avec les données de l'exception : la ligne de début et de fin, le type d'erreur rencontrée, la phase de l'analyse durant laquelle l'exception a été levée...

Côté Web, on peut parser la chaîne de caractères et récupérer le JSON parsé pour manipuler les informations sur l'erreur rencontrée (voir Figure 4). CodeMirror offre des fonctions permettant de manipuler l'état global de l'éditeur : son thème, son contenu, le langage de la coloration syntaxique, les options activées... On y retrouve surtout la liste des lignes qui composent l'éditeur. On peut alors leur affecter des changements de style pour les mettre en lumière ou faire apparaître

```
{
    "loc": {
        "beginning": { "line": 10, "column": 19},
        "end": {"line": 10, "column": 21}
    },
    "info": "Error on line 10 col 19: Syntax error."
}
```

FIGURE 6 – Sortie d'erreur Ocaml parsée

une infobulle avec le message d'erreur si on passe la souris dessus...

On offre ainsi les outils nécessaires à l'utilisateur pour bien écrire et déboguer son code MiniML. Néanmoins, il faut encore compléter la *pipeline* en traitant toutes les sorties possibles d'Autobill, notamment les équations sur les contraintes mémoires du programme analysée qu'il va falloir résoudre.

2.1.3 La résolution de contraintes

Autobill exprime les contraintes mémoires qui s'appliquent à un programme dans des formats proches d'équations mathématiques. Il propose de type de sortie : une sous Coq et une sous MiniZinc. Nous nous concentrerons principalement sur MiniZinc dans cette partie, Coq n'étant pas encore totalement opérationnel et demandant des ressources qui dépassent le cadre du Master d'informatique.

MiniZinc est un langage permettant de décrire des problèmes de manière déclarative à l'aide de contraintes logiques. Il fournit des éléments de syntaxe et de notations très similaires à la programmation et aux mathématiques pour rédiger des modèles qui, lorsqu'ils seront passés à des compilateurs et des solveurs dédiés, vont répondre à des problématiques d'optimisation ou de satisfiabilité. Dans notre cas, on s'intéresse à l'optimisation : on veut déterminer les bornes mémoires minimum du programme passée à Autobill. Néanmoins, on rencontre un problème similaire qu'avec Autobill : la librairie est codée en C++, langage compilé, incompatible avec l'environnement Web. Là encore, l'option de WebAssembly est envisageable avec un compilateur comme Emscripten qui supporte la traduction de code C/C++ vers bytecode WebAssembly. Ici, la librairie core de MiniZinc est beaucoup plus fournie et ne contient pas un point d'entrée direct explicite, comme avec le binaire d'Autobill. La compilation devient alors une tâche plus compliquée et la manipulation sûre de l'outil n'est pas non plus garantie par la suite.

```
        int: n;
        var 1..n: x; var 1..n: y;
        constraint x+y > n;
        solve satisfy;
```

FIGURE 7 – Exemple de modèle en MiniZinc

```
    const model = new Model()
    model.addFile("playground.mzn", code)
    const solve = await model.solve({
      options: {
        solver: "gecode",
      },
    })
```

2.2 Serveur + client

Dans le stade actuel d'Autobill, une architecture avec un client seul peut répondre aux exigences du projet STL. Néanmoins, au fur et à mesure que le projet avançait, nous nous sommes rendu compte d'un problème. Autobill évolue constamment et rien ne garantit que ses itérations suivantes puissent être supportées par notre solution. Par exemple, la utilisation de la librairie core de MiniZinc peut échouer en mode client unique. 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 MiniML

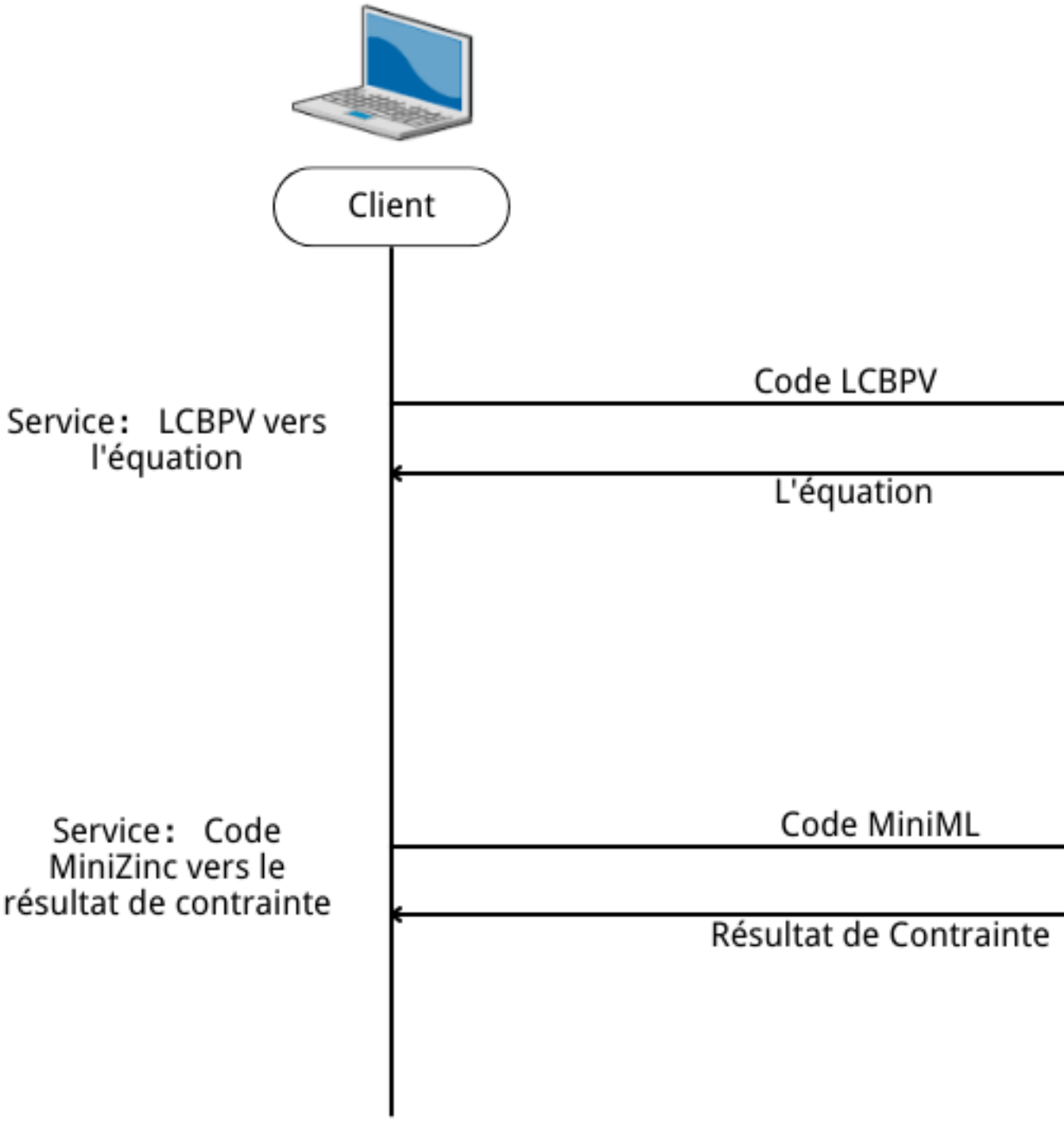

FIGURE 8 – Schéma de communication

```
    const fs = require('fs');
    const tmpFile = './temp.mzn';
    fs.writeFileSync(tmpFile, code);
    const cmd = `minizinc --solver Gecode ${tmpFile}`;
    exec(cmd, (error, stdout, stderr) => {
        fs.unlink('temp.mzn', (err) => {
            if (err) throw err;
        });
    }
```

FIGURE 9 – Exécution de code Minizinc

Nous vouslons diviser notre architecture Client-Server en deux partie, la partie MiniML et la partie résolution de contraintes. Dans la partie MiniML, nous avons mis en place 5 services principaux en utilisant le protocole HTTP (voir Figure 8) : *'/MiniML/toMiniMLAST'*, *'/MiniML/toLCBPV'*, *'/MiniML/toAutobill'*, *'/MiniML/toAutobillType'* et *'/MiniML/toEquation'*.
Pour le service *'/MiniML/toMiniMLAST'*, le client envoit le code du MiniML au server via une rêquete POST. Le server le transforme vers l'AST de MiniML et le renvoit au client.
Pour le service *'/MiniML/toLCBPV'*,le client envoit le code du MiniML au server via une rêquete POST. Le server le transforme vers l'AST de LCBPV et le renvoit au client.
Pour le service *'/MiniML/toAutobill'*, le client envoit le code du MiniML au server via une rêquete POST. Le server le transforme vers le code d'Autobill et le renvoit au client.
Pour le service *'/MiniML/toAutobillType'*, le client envoit le code du MiniML au server via une rêquete POST. Le server le transforme vers le code d'Autobill typé et le renvoit au client.
Pour le service *'/MiniML/toAutobillType'*, le client envoit le code du MiniML au server via une rêquete POST. Le server le transforme vers l'équation résultant de l'analyse statique et le renvoit au client.

2.2.2 La résolution de contraintes

Dans la partie résolution de contraintes, nous avons un service *'/minizinc/gecode'* qui éxecute le code MiniZinc. Le client envoie le code de MiniZinc au serveur via une requête POST. Le serveur écrit le code dans un fichier temporaire puis l'exécuter à l'aide de l'exécutable de ligne de commande minizinc avec le solveur Gecode (voir Figure.9), et renvoie le résultat au client.

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 [3].

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** [23] : *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 [24] pour toute la partie MiniML et le processus de transformation vers CBPV. Puis, les cours de programmation concurrente répartie, réactive et réticulaire [22], 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 [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
- 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**