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 [1].
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.

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

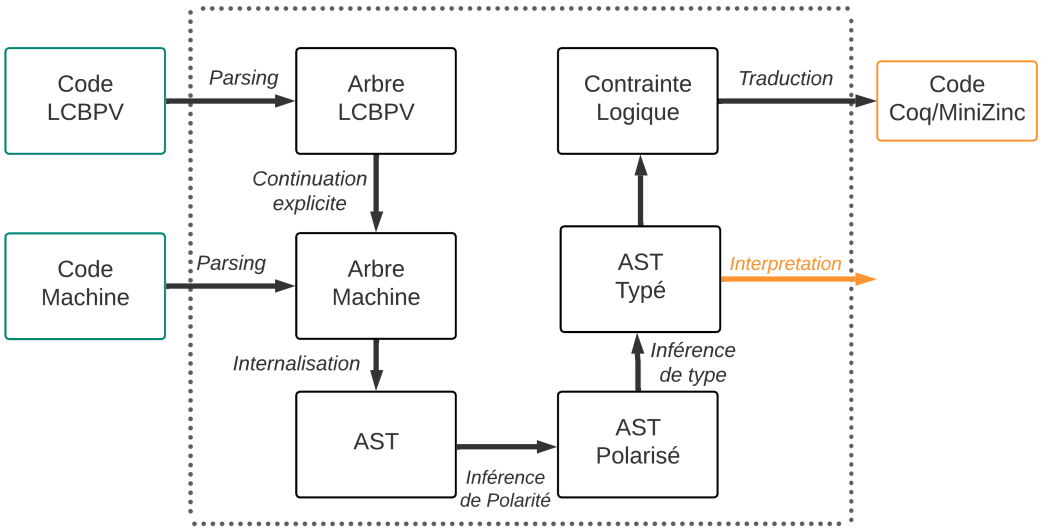

FIGURE 1 – Représentation simplifiée d'Autobill

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 1, 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 1 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.

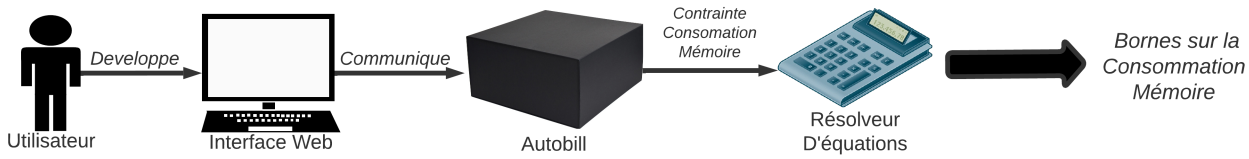

FIGURE 2 – Représentation du système cible

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 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

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.

Enfin, une dernière piste suggérée a été l'utilisation de langages de programmation camélien

```
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

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 afin de les intégrer dans notre projet.

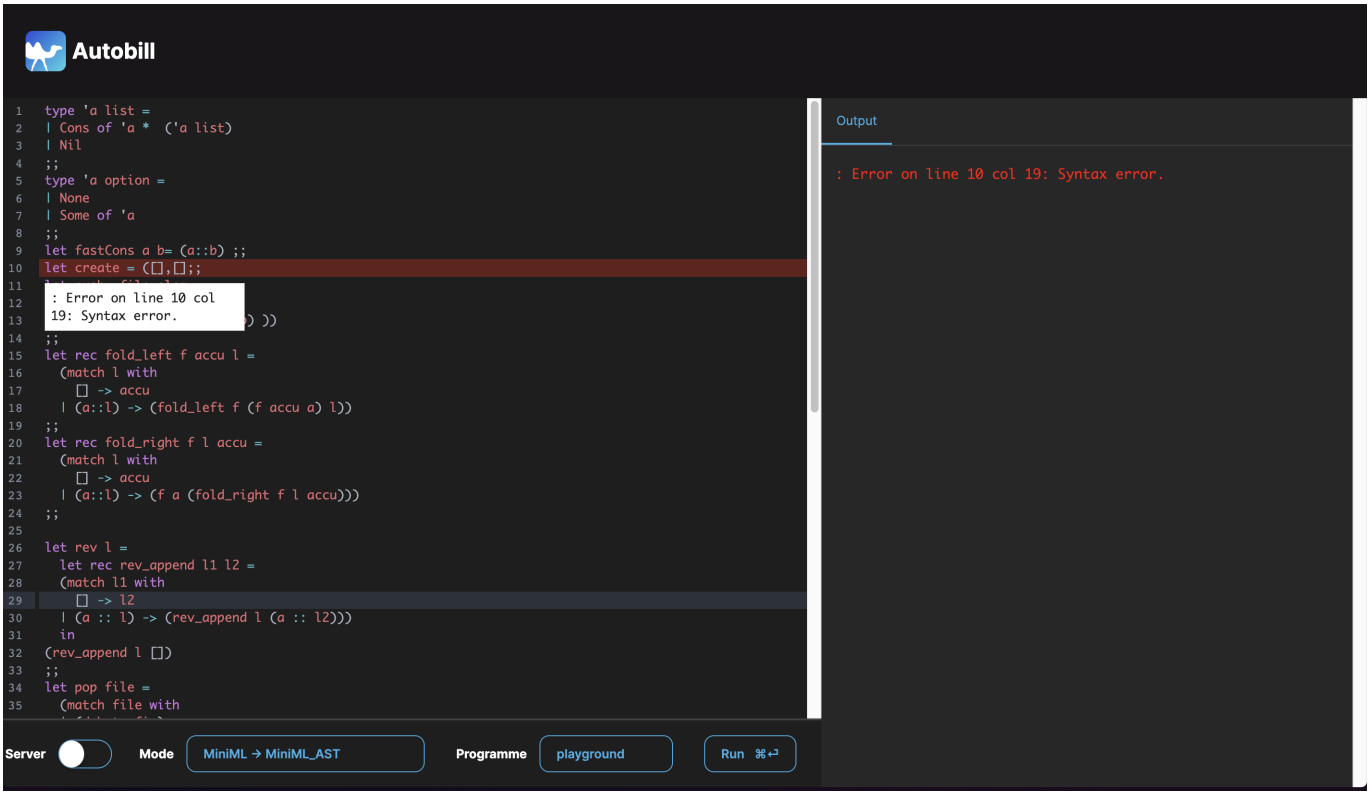

FIGURE 5 – Interface pour Autobill, dans le scénario d'une erreur

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

```
{
    "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

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

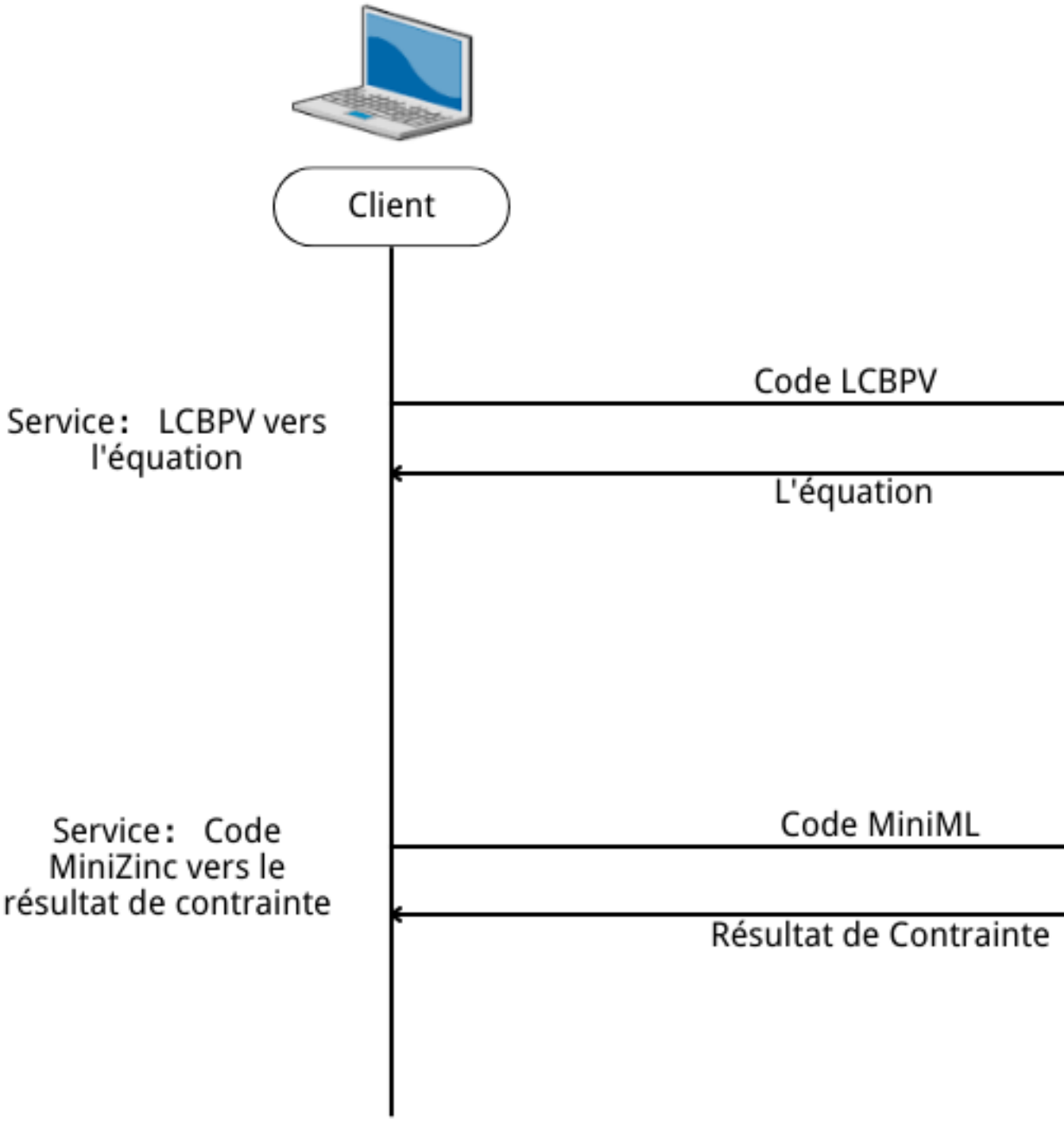

FIGURE 8 – Schéma de communication

```
    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**. Nous avons pris la décision de rendre la syntaxe MiniML parfaitement compatible avec OCaml simplifiant les comparaisons avec d'autres outils utilisant **OCaml** en entrée.

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

3.2 Grammaire

Voici la grammaire BNF du language MiniML. Pour des raisons de lisibilité, tout les éléments de sucre syntaxique ne seront pas détaillés ici. Les extensions non compatibles avec Ocaml mais compatible avec **Autobill** sont en Orange.

Elements Nommés

<Id> ::= ['a - z' 'A - Z' '0 - 9' '_']+
<ConstructeurId> ::= ['A - Z'] <Id>
<Vartype> ::= ''' ['a - z'] ['0 - 9']*

Programmes

<Prog> ::= | <Expr>
| <Def> ';;' <Prog>

Definitions

<Def> ::= | 'let' <Id> '=' <Expr>
| 'type' <Vartype>* <Id> '=' <NewContructors>

<NewContructors> ::= | <ConstructeurId> 'of' <Type>
| <NewContructors> '|' <NewContructors>
Debut Extension | <ConstructeurId> 'of' <Type> 'with' <ParamAssigns>

<ParamAssigns> ::= | <VarType> '=' <ParmExpr>
| <ParamAssigns> ',' <ParamAssigns>
<ParmExpr> ::= | '1'

| | <VarType>
| | <ParmExpr> '+' <ParmExpr>
Fin Extension | | <ParmExpr> '*' <ParmExpr>

Expressions

<Litteral> ::= | ['0 - 9']+
| ['true' | 'false']
| '(' ')'

<Expr> ::= | <Litteral>
| <Id>
| <UnaryOperator> <Expr>
| <Expr> <BinaryOperator> <Expr>
| <Expr> <Expr>
| <Expr> ',' <Expr>
| 'let' <Id> '=' <Expr> 'in' <Expr>
| 'fun' <Id> '->' <Expr>
| 'fun' 'rec' <Id> <Id> '->' <Expr>
| <ConstructeurId> <Expr>
| 'match' <Expr> 'with' <MatchCase>

<UnaryOperator> ::= 'not'
<BinaryOperator> ::= ['and' | 'or' | '+' | '-' | '/' | '%' | '*' | '<' | '>' | '=']

Filtrage et Motifs

<MatchCase> ::= | <Pattern> '->' <Expr>
| <MatchCase> '|' <MatchCase>

<Pattern> ::= | <Litteral>
| <Id>
| <Pattern> ',' <Pattern>
| <ConstructeurId> <Pattern>

Types

```
<Type>  ::=  |  <Vartype>
             |  <Id>
             |  <Type> <Type>
             |  <Type> '->' <Type>
             |  <Type> '*' <Type>
```

3.3 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*.

En CBPV, les valeurs et les calculs sont deux notion bien distincte, les calculs peuvent produire et donc être réduit on dit qu'ils "font", tandis que les valeurs sont des contenants et ne peuvent pas être réduit on dit qu'ils "sont". Lorsqu'on effectue la traduction depuis une autre sémantique, nous devons donc décider comment et quand nos calculs vont être effectuer.

Dans le cadre de ce projet, 3 constructions de LCBPV nous intéressent particulièrement, les **Closure**, les **Thunks**.

- Les **Thunks** sont des expressions non evaluée, ils sont utilisés pour représenter un calcul qui n'a pas encore été effectué. On peut donc les voir comme des lambdas sans arguments. Pour forcer l'évaluation d'un **Thunk** il suffit d'utiliser l'instruction **force**.
- Les **Closures** qui permettent de capturer un environnement et une expression et de les lier.
- Les **Methods** qui dans notre cas permettent de définir des calculs à effectuer avec des arguments.

Pour plus d'information sur LCBPV sa syntaxe, sa sémantique et ses constructions , vous pouvez vous référer à l'annexe de ce rapport.

3.4 Semantique de traduction

Puisque MiniML est un language *Call By Value*,ici nous décidons que toute traduction nécessitant le passage par la notion de calculs dans CBPV forcera immédiatement l'évaluation de ceux-ci. Permettant ainsi une equivalence parfaite entre tout comportement de MiniML traduit en LCBPV.

Pour décrire la traduction de MiniML en LCBPV, nous allons opté pour une traduction de syntaxe concrète à syntaxe concrète. Vous trouverez ci-dessous toutes les cas d'intérêts de la traduction.

Notation

$[\![let\ v\ =\ e_1\ in\ e_2]\!]_{Expr} \rightarrow let\ v =\ [\![e_1]\!]\ in\ [\![e_2]\!]$

- $[\![\_]\!]_{Expr} \rightarrow$ est la traduction d'un noeud expr du langage MiniML vers le langage LCBPV

  - À l'interieur des $[\![]\!]$, nous trouvons les elements propre au langage MiniML qui sont traduit vers le langage LCBPV.
  - À l'exterieur des $[\![]\!]$, nous trouvons les elements propre au langage LCBPV.
- X_n est le n-ième sous-noeud de l'arbre de syntaxe abstrait.
  Selon les cas X peut être :
  - e pour les expressions
  - p pour les motifs
  - t pour les types
  - d pour les définitions
    - dc pour les définitions de constructeurs
  - v pour les elements nommées
  - op pour les opérateurs
  - l pour les litteraux
- $X_1 \dots X_n$ est la liste des sous-noeuds de type X allant de 1 à n

Programmes

Un programme MiniML est une suite de taille arbitraire de définitions suivie d'une expression.

$$[\![d_1 \dots d_n\, e]\!]_{Prog} \to [\![d_1 \dots d_n]\!]\ return\ [\![e]\!]$$

Définitions

On définit l'operation de traduction $[\![\_]\!]_{Def}$ selon les cas de construction des définitions précisées par la règle de grammaire : **<Def>**

On distingue deux noeuds de types **<Def>** :

- Les définitions de **Variables Globales**,
- Les définitions de **Types**.

$$\begin{aligned} &\text{(VARDEF)} \quad [\![let\, v\, =\, e]\!]_{Def} \to let\, v\, =\, [\![e]\!] \\ &\text{(TYPDEF)} \quad [\![type\ v_1 \dots v_N\, =\, dc_1 \dots dc_N]\!]_{Def} \\ &\qquad\qquad \to data[\![v_1]\!]([\![\dots v_N]\!] : +) = [\![dc_1 \dots dc_N]\!] \end{aligned}$$

Définitions de Constructeurs

On définit l'operation de traduction $[\![\_]\!]_{DefContructors}$ selon les cas de construction précisées par la règle de grammaire : **<NewContructors>**

On distingue deux noeuds de types **<NewContructors>** :

- Les définitions de **Constructeurs Classiques** qui sont compatibles avec OcamL.
- Les définitions de **Constructeurs Equationnels** qui sont une extension spécifique pour **Autobill** incompatible avec OcamL et dont la traduction n'est pas encore totalement définie.

$$\text{(MLCONSTRUCTDEF)} \quad [\![v\ of\ t]\!]_{DefContructors} \to v[\![t]\!]$$

Types

On définit l'operation de traduction $[\![\_]\!]_{Type}$ selon les cas de construction des types précisées par la règle de grammaire : **<Type>**

On distingue 4 noeuds de types $Type$:

— Les **Variables de types**.
— Les **Applications de types**.
— Les **Lambda**.
— Les **Tuples**.

$$
\begin{aligned}
\text{(TVAR)}\ & [\![v]\!]_{Type} \rightarrow v \\
\text{(TAPP)}\ & [\![t_1 t_2]\!]_{Type} \rightarrow [\![t_1]\!]\ [\![t_2]\!] \\
\text{(TCLOS)}\ & [\![t_1 \text{->} t_2]\!]_{Type} \rightarrow Exp(Fun[\![t_1]\!] \text{->} Thunk[\![t_2]\!]) \\
\text{(TTUPLE)}\ & [\![t_1 * t_2]\!]_{Type} \rightarrow [\![t_1]\!]\ [\![t_2]\!]
\end{aligned}
$$

Expressions

On définit l'operation de traduction $[\![\_]\!]_{Expr}$ selon les cas de construction des expressions précisées par la règle de grammaire : **<Expr>**

On distingue 8 noeuds d'intérêts de types **<Expr>** :

— Les **Tuples**.
— Les **Appels de fonctions**.
— Les **Fixation**.
— Les **Lambda**.
— Les **Fonctions Recursive**.
— Les **Constructions**.
— Les **Correspondance de motifs**.
— Les **Appels d'operateurs**.

Note : On ne prend pas en compte les **Litteraux** et les **Variables** dans ce descriptif car leurs traductions sont directes.

$$
\begin{aligned}
\text{(BINARY)}\ & [\![e_1\ op\ e_2]\!]_{Expr} \rightarrow [\![e_1]\!]\ op\ [\![e_2]\!] \\
\text{(UNARY)}\ & [\![op\ e]\!]_{Expr} \rightarrow op\ [\![e]\!] \\
\text{(TUPLE)}\ & [\![e_1\ ,\ e_2]\!]_{Expr} \rightarrow \text{Tuple}([\![e_1]\!]\ [\![e_2]\!]) \\
\text{(LAMBDA)}\ & [\![fun\ v\ \rightarrow\ e]\!]_{Expr} \rightarrow exp(\ get \mid call(v) \text{->} thunk[\![e]\!]) \\
\text{(FUN REC)}\ & [\![fun\ rec\ v_1\ v_2 \rightarrow\ e]\!]_{Expr} \rightarrow exp(\ rec\ v_1\ is\ get \mid call(v_2) \text{->} thunk[\![e]\!]) \\
\text{(CALL)}\ & [\![e_1 e_2]\!]_{Expr} \rightarrow \{ \\
& \qquad open\ exp\ v_1 = [\![e_1]\!] \\
& \qquad force\ thunk(v_2) = (v_1).call[\![e_2]\!] \\
& \qquad return\ v_2 \\
& \} \\
\text{(BIND)}\ & [\![let\ v\ =\ e_1\ in\ e_2]\!]_{Expr} \rightarrow \{ \\
& \qquad let\ v = [\![e_1]\!] \\
& \qquad return\ [\![e_2]\!] \\
& \}
\end{aligned}
$$

$$\text{(CONSTRUCT)} \quad [\![v\, e]\!]_{Expr} \rightarrow v[\![e]\!]$$
$$\text{(MATCH)} \quad [\![match\, e\, with\, p_1 \ldots p_N]\!]_{Expr} \rightarrow match\, [\![e]\!]\, with\, [\![p_1]\!] \ldots [\![p_N]\!]\, end$$

Les cas les plus intéressants sont les cas de **Fonctions** et les **Appels de Fonctions**. La traduction d'une fonction ce déroule ainsi :

1. A l'aide de l'instruction **get** on crée une methode **call** qui prend en paramètre un argument.
2. Ensuite dans le corps de cette methode on crée un **Thunk** qui contient la traduction du corps de la fonction.
3. Puis on crée une **Closure** qui contient la methode ainsi créée et son environnement.

Au vu de la traduction des fonctions, on peut en déduire la traduction des appels :

1. On ouvre la **Closure** de la fonction. Pour récupérer la méthode **call** et l'environnement.
2. On applique la méthode **call** sur l'argument fourni au moment de l'appel. Pour récupérer un **Thunk** sur le corps de la fonction avec l'environnement enrichi de l'argument.
3. On force le **Thunk** ainsi obtenu pour avoir l'evalution immédiate du corps de la fonction.

Motifs et Filtrage

On définit l'operation de traduction $[\![\_]\!]_{Case}$ selon les cas de construction des expressions précisées par la règle de grammaire : <MatchCase>
On distingue 4 noeuds d'intérêt de types <MatchCase> :
— Les patterns sur **Litteraux**.
— Les patterns sur **Variables**.
— Les patterns sur **Tuple**.
— Les patterns sur **Constructeurs**.

$$\text{(PATTERN LIT)} \quad [\![l \text{ -> } e]\!]_{MatchCase} \rightarrow [\![l]\!] \text{ -> } [\![e]\!]$$
$$\text{(PATTERN VAR)} \quad [\![v \text{ -> } e]\!]_{MatchCase} \rightarrow v \text{ -> } [\![e]\!]$$
$$\text{(PATTERN TUPLE)} \quad [\![v_1 \text{ , } v_2 \text{ -> } e]\!]_{MatchCase} \rightarrow Tuple(v_1\ v_2) \text{ -> } [\![e]\!]$$
$$\text{(PATTERN CONSTR)} \quad [\![v_1\, v_2 \text{ -> } e]\!]_{MatchCase} \rightarrow v_1\, v_2 \text{ -> } [\![e]\!]$$

On notera que la traduction des patterns ne prend pas en compte les patterns profonds, c'est à dire les patterns imbriqués dans d'autres patterns. En effet, les patterns profonds ne sont pas encore supportés par le language cible.

3.5 Implémentation

Dans le cadre de ce projet MiniML, dispose d'une implémentation écrite en **OCaml** nous avons utilisé ce langage pour deux raisons principales. Permmettre une meilleure intégration avec autobill qui est écrit en OCaml et permettre une compatibilité avec les deux architectures du projet. Cela nous a permis d'effectuer la traduction non pas de syntaxe concrète à syntaxe concrète mais de syntaxe abstraite à syntaxe abstraite. Nous epargnant ainsi une nouvelle étape de parsing. Une autre divergence avec la spécification est que nous avons utilisé une traduction recursive terminale par soucis d'optmisation.

4 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 [7] pour toute la partie MiniML et le processus de transformation vers CBPV. Puis, les cours de programmation concurrente répartie, réactive et réticulaire [8], 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.

Références

[1] H. Suzanne, "Autobill." https://gitlab.lip6.fr/suzanneh/autobill, 2023.

[2] J. Hoffmann and S. Jost, "Two decades of automatic amortized resource analysis," *Mathematical Structures in Computer Science*, vol. 32, pp. 729–759, Mar. 2022.

[3] J. Hoffmann, "Resource aware ml." https://www.raml.co/, 2023.

[4] P. B. Levy, "Thése sur call-by-push-value." https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf, 2001.

[5] "Minizinc." https://www.minizinc.org/, 2023.

[6] INRIA, "The coq proof assistant." https://coq.inria.fr/, 2023.

[7] P. MANOURY, "Cours aps." https://www-apr.lip6.fr/~manoury/Enseignement/2021-22/APS/index.html, 2021.

[8] E. Chailloux, "Cours pc2r." https://www-apr.lip6.fr/~chaillou/Public/enseignement/2018-2019/pc2r/public/cours/Cours08.pdf, 2018.

Annexe

Syntaxe et sémantique informelle de LCBPV

Call-By-Push-Value En Bref

Le langage λ-CBPV ("lambda-call-by-push-value", ou juste LCBPV pour les intimes) est une réprésentation intermédiaire de haut niveau, dans laquel on peut compiler les langages à la ML (avec évaluation stricte), ou à la Haskell (avec évaluation paresseuse), ou mélanger les deux pour compiler des fonctionalités avancées (comme certaines concept de programmation objets).

Dans LCBPV, on a des valeurs et des calculs. Les valeurs *sont* des choses mais de font rien, les calculs *font* des choses mais ne contiennent pas de données. On a donc deux *sortes* de type : si `T` est type de valeur, on le défini comme `T : +` , et de même, `T : -` est un type de calcul. Pour aider la compréhension, quand `T` est un type de valeur, on le notera parfois `T+`, et `T-` si c'est un type de calcul.

Les types de valeurs

les entiers Classique. On les écrits en base 10, avec un '-' devant pour les négatifs. On dispose des opérations classiques : addition, substraction, division euclidienne, reste, et négation. On peut tester pour l'égalité et l'ordre. Le type de valeur des entier est `Int : +`

```
0 1 2 3530530899 -2 -4343  (littéraux)
1+1 2-2 3*3 4*4 -5         (opérations)
6==6 7<7 8=<8 (tests)
```

Les booléens Encore classique. `true`, `false`, `and`, `or` et `not`. Le type de valeur des booléens est `Bool : +`

```
true false
&& || !
```

On a un if/then/else :

```
if 1>0 then 45 else 422
```

Les tuples On les note `tuple(e_1, ..., e_n)`, et ils ont le type `Tuple(T_1+, ..., T_n+)`. Chacun des `n` composant est une valeur `e_i : T_i : +`. Le 0-uplet/tuple vide est noté `unit()` de type de valeur`Unit`. On accède aux composants `e_1`,...,`e_n` des tuples par pattern matching :

```
match e_tuple with
 | tuple(x_1, ..., x_n) -> ...
end
```

Les types sommes Les sommes binaires on le type de valeur `Sum(T+,U+) : +`. Ces deux constructeurs sont `inj{1,2} : T -> sum(T,U)` et `inj{2,2} : U+ -> Sum(T+,U+)`. En général, on peut faire des sommes avec n types `T_1+, ..., T_n+`. On note le type de valeurs correspondant `Sum(T_1+, ...,T_n+) : +` et on a des constructeurs `inj{i,n}` pour tout i entre 1 et n. On peut pattern matcher sur les sommes :

```
match e_sum with
  | inj{1,n}(x) -> ...
  | inj{2,n}(x) -> ...
  ...
  | inj{n,n} -> ...
end
```

Définir des types de valeurs On peut, comme en OCaml, définir des types par cas. Par exemple, le type des listes et le type option sont définis de la manière suivante :

```
data List(A : +) =
  | nil()
  | cons(A, List(A));

data Option(A : +) =
  | none()
  | some(A);
```

On peut alors créer des listes en utilisant les constructeurs `cons` et `nil`, par exemple, la liste [1,2,3] sera encodée par `cons(1,cons(3,cons(3,nil)))`. On peut aussi faire du fitrage par motif sur les listes. Par exemple, si on a une liste d'entiers `l`, l'expression ci-dessous renvoit la tête de la liste dans un type option.

```
match l with
  | nil() -> {return none()}
  | cons(h,t) -> {return some(h)}
end
```

Les blocs, thunks, et fermetures

Blocs LCBPV utilise des blocs pour séquencer les opérations. Un bloc est une séquence d'instruction, séparées par un point-virgule, et terminé par un `return e` qui évalue `e` et renvoie et renvoie la valeur correspondante. On a trois instructions : `let open force`. La première est `let x = e;`, elle évalue `e`, et affecte le résulat à `x`. `x` est en portée dans les instruction suivantes dans le bloc et dans l'expression de retour. Par exemple :

```
let x = 2+2;
let y = 3+1;
return x + y
```

On peut remplacer une expression par bloc avec des allocades. C'est à dire, les expressions `e`, `{return e}`, et `{let x = e; return x}` sont interchangeables. On va maintenant voir `force` et `open`, qui gèrent les *thunks* et les *fermetures*.

Les thunks Quand un calcul renvoie une valeur, il la renvoi encapsulée dans un calcul nommé *thunk* et noté `thunk(e)`, où est une expression quelqueonque qui renvoie une valeur. Il faut *forcer* le thunk avec `force x = thunk(e)` pour calculer effectivement la valeur de `e`, et l'affecter à `x`. Pour l'exemple, comparez les deux fragments de code suivants, qui calculent différement le même résultat. Dans le premier, on calcule 2+2 avant de calculer 3+1, et dans le second on calcule 3+1 en premier et 2+2 après.

```
// fragment 1:
let x = 2+2;
let y = 3+1;
return x + y

//fragment 2:
let x = thunk(2+2);
let y = thunk(3+1);
force thunk(y') = y;
force thunk(x') = x;
return x'+y'
```

Si une expression `e : T : +` renvoie une valeur de type de valeur `T`, alors le thunk `thunk(e) : Thunk(T) : -` est une valeur de type de calcul `Thunk(T)`

Les fermetures Comme les thunks qui sont des calculent qui renvoient une valeur après évaluation, on peut créer des valeurs qui stockent un calcul avant son évaluation. si on a un calcul `f`, on peut le mettre dans une *fermeture* `closure(f)` qui bloque son évaluation et stocke les valeurs de ses variables libres. Quand on *ouvre* la fermeture avec `open g = closure(f)`, on assigne `g=f` et on remet les valeurs dont `f` dépend sur la pile. On peut alors accéder à `g` correctment (sans segfault).

Par exemple, dans le code suivant, un créé une fermeture sur un calcul qui renvoi un entier. Comme les entiers sont des valeurs, on créé une fermeture autour d'un trunk qui renvoi un entier. On voit que quand on définit un calcul, on peut évaluer les calculs en plusieurs morceaux : quand ils sont définis, quand on ouvre leur fermeture, et quand on les force.

```
// Au début, je n'ai pas encore évalué ‘a=2+2‘ ni ‘b=3+1‘
let f = {

  let a = 2+2;
  // Ici, j'ai évalué 'a' mais pas ‘b‘

  // la fermeture capture la variable ‘a=4‘ qui est libre dans son corps
  return closure({

      // ce bloc est bloqué par la fermeture
      let b = 3+1;
      return thunk(a+b)
  })
};
```

```
// Ici, je n'ai toutjours pas évalué ‘b‘
open closure(f') = f;
// Après l'ouverture de ‘f‘, ‘b‘ est évalué et on a assigné ‘f’ = thunk(a+b)‘

// Puis on force le thunk pour évaluer ‘a+b‘ et récupérer 8
force thunk(x) = f';
return x
```

Si `e : T : -` est une expression qui renvoie un calcul de type `T`, alors `closure(e) = Closure(T) : +` est une valeur de type de valeur `Closure(T)`

Les autres types de calculs

Les fonctions Les fonctions en LCBPV sont des calculs. Comme en Java ou C++, on les représente par des "objets" avec une méthode `call`. Par exemple, la fonction "f : x $\to$ 2x+3" sera écrite et appelée de la manière suivante. Notons que les fonctions qui renvoient des types de valeurs doivent encapsuler leur valeur de retour dans un thunk, et qu'il faut forcer le thunk pour terminer l'appel.

```
// ‘y‘ est défini comme un calcul, avec une méthode ‘call‘:
let f = get
    | call(x) -> thunk(2 * x + 3)
  end;

// ‘y‘ est un thunk qui calcule "f(1)". On a pas encore évalué f !
let y = f.call(1);

// Ici on calcule l'appel "f(1)", qui renvoie 2*1+3 = 5
force thunk(z) = y;

return z // on renvoie z = 5
```

Le type des fonctions est `Fun(T_1+, ..., T_n+) -> T-`, avec chaque argument un type de valeur et le résultat un type de calcul. Les fonctions en LCBPV ne sont pas currifiés. Une fontion de type `Fun(A+, B+) -> C-` doit recevoir ces deux arguments simultanément. Cette fonction, currifiée, serait de type `Fun(A+) -> Fun(B+) -> C-`, ou encore `Fun(A+) -> Thunk(Closure(Fun(B+) -> C-))`.

Les "paires paresseuses" Les paires paresseuses sont des objets avec plusieurs méthodes qui renvoient chacune un type différent. Au final, seul(s) le(s) élément(s) accédé(s) sont effectivement calculées, d'où le coté paresseux. Si on a une paire paresseuse `p` avec n, éléments, on appele la méthode `p.proj{i,n}()` pour sélectionner le i-ème. Notons que qu'aucune des méthodes ne prend d'arguments, et qu'il faut forcer le thunk pour effectivement calculer le i-ème élément.

On défini une paire paresseuse en spécifiant chacune des méthodes `proj{i,n}` dans une clause `get ... end` comme pour les functions :

```
let douzaines = get
  | proj{1,3}() -> 12
  | proj{2,3}() -> 24
  | proj{3,3}() -> 36
end;

force thunk(x) = douzaine.proj{1,3}();

return x // renvoie 12
```

Définir des types de calculs On peut aussi définir des types de calcul au cas-par-cas, c'est à dire en définissant chaque méthode. Par exemple, on peut définir le type des itérateurs sur les listes de la manière suivante : un itérateur est un objet avec deux méthodes, une qui consome un élément de liste et renoie un nouvel itérateur, et une second qui termine l'itération quand on croise `nil()`. On définit alors un type `Iter(A, B)` qui consome des listes de type `A` et renvoi un `B`. La méthode `use_nil()` renvoie une thunk sur le `B` final, et `use_cons` prend un `A` et renvoi un `Iter(A,B)`. Notons que comme `Iter(A,B) : -` est un type de calcul, on a pas besoin de renvoyer un thunk dans `use_cons`.

```
comput Iter(A : +, B : +) =
  | use_nil() -> Thunk(B)
  | use_cons(A) -> Iter(A,B);
```

On définit souvent les itérateurs par récurrence, et on vera cela plus bas.

Le reste des fonctionalités

Définitions récursives On peut définir des calculs récursifs avec le constructeur spécial `rec x is ...`. Par exemple, en OCaml on écrirai la fonction factorielle :

```
let rec fact n =
  if n = 0 then
    1
  else
    n * (fact (n-1))
```

On écrirai globalement la même chose en LCBPV, mais en séparent le `let` et le `rec` : on définit `let fact = rec self is ...`. `self` est alors une fermeture sur la fonction récursive, et est en portée dans la définition de `f`. Formellement, `fact` doit avoir un type de calcul `T-`, et `self` sera de type `Closure(T-)`. Il est essentiel que `self` soit une fermeture pour des raisons techniques, et donc doit être d'un type différent de `f`.

Par exemple, on définit une function récursive `fact` de type `Fun(Int)->Thunk(Int)`. Dans la définition, `self` est une fermeture `Closure(Fun(Int) -> Thunk(Int))`, et on l'ouvre pour retrouver `fact`.

```
let fact = rec self is
    get
    | call(n) ->
         if n==0 then
           thunk(1)
         else {
           open f = self;
           force m = f.call(n-1);
           return thunk(m * n)
         }
    end
```

Déclarations Pour simuler des bibliothèques, on peut déclarer des types et des valeurs au toplevel sans les définir. Quand on déclare une valeur, on doit spécifier son type, et quand on déclare un type, on doit déclarer la sorte (valeur/calcul) de ces argument et la sort du type. Par exemple, pour déclarer les tableaux :

```
decl type Array : (+) -> +
decl array_init : Fun(Int, A) -> Thunk(Array(A))
decl array_read : Fun(Array(A), Int) -> Thunk(Option(A))
decl array_write : Fun(Array(A), A, Int) -> Thunk(Unit)
```

Notons que la variable de type `A` n'est pas déclarée, c'est implicitement une variable de type polymorphique. La fonction `array_read` aurait en OCaml le type `'a. 'a array * int -> 'a option thunk`.

Types linéaires Finalement, LCBPV supporte les types linéaires. C'est à dire que les variables ne peuvent être utilisées qu'une fois ni plus ni moins dans chaque execution du programme. Pour pouvoir interpréter les languages normaux (sans types linéaires), on introduit des *fermetures partageables*.

Les fermetures partagées fonctionnent exactement comme les fermetures : - Si `e : T-` est une expression de calcul, `exp(e) : Exp(T-)` est une valeur de tpye `Exp(T-)`, comme pour les fermetures avec `closure` et le constructeur de type `Closure`. - On peut accéder le caclul sous-jaçant avec l'instruction `open exp(x) = e;`, qui évalue le calcul dans la fermeture et l'assigne à `x`.

Avec les types linéaires, on a deux différences : - On doit utiliser toutes les variables une fois, ni plus *ni moins*. - On peut néanmoins partager les `exp` librement, c'est à dire les utiliser zéro ou plus de une fois. - On peut créer des `closure(e)` pour n'importe quel `e : T-`, mais pas pour `exp`. Si on veut mettre une expression `e : T-` dans une fermeture partagée `exp(e)`, il faut que les variable libres dans `e` soient toutes de type `Exp(...)`. Donc, les fermetures partagées ne peuvent dépendre que d'autres fermetures partagées, ou ne pas avoir de variables libres.

Tout les tokens

— `x` un nom de variable,
— `litt` est un littéral `true`, `false`, ou un des entiers
— `op` est une opération primitive sur les entiers et les booléens (voir plus haut).
— `k` un nom de constructeur, dont `unit`, `tuple` et `inj{i,n}`.
— `m` un nom de méthode, dont `call` et `proj{i,n}`
— `A` un nom de variable de type, dont : `Int Bool Unit Zero Top Bottom`
— `K` un nom de constructeur de type, dont : `Tupple Sum Choice Thunk Closure Exp`.
— Le constructeur de type `Fun` avec syntaxe spécifique.
— plus les symboles et mots-clés de la syntaxe si dessous.

Syntaxe formelle

Notes : - dans `match`, `get`, `data` et `comput`, les cas sont séparés par des `|`, et le premier est optionel. On peut donc écrire `match e with k() -> ... | ... end` pour `match e with | k() -> ... | ... end`.

```
T ::= T+ | T- | A | K(T_1, ..., T_n) | Fun(T_1, ..., T_n) -> T'


x ::= x | x : T


expr ::=

  x                       (variables)

  litt                    (littéraux)

  op(e, ..., e)           (primitives)

  e : T                   (expression avec une annotation de type)

  k(e_1, ..., e_n)        (constructeur (comme Some, Cons, Nil, etc.))

  thunk(e)                (thunks renvoyant une valeur ‘e‘)

  closure(e)              (fermeture sur un calcul ‘e‘)

  exp(e)                  (fermeture partagée)

  e.m(e_1, ..., e_n)   (appel de méthode sur un calcul (à utiliser avec ‘get ... end‘))

  if e then e else e      (conditionelle)

  match e with            (pattern matching (à utiliser avec ‘e‘ = ‘k(...)‘))
    | k_1(x_1, ..., x_n) -> e_1
```

```
    | k_2(y_1, ..., y_n) -> e_2
    ...
  end

  get                    (définition de calcul)
    | m_1(x_1, ..., x_n) -> e_1
    | m_2(y_1, ..., y_n) -> e_2
    ...
  end

  { block }              (exécution de bloc)

  rec x is e             (calculs récursifs)

  absurd(e)              (indique que le code ‘e‘ est mort (jamais accessible au runtime))


block ::=

  return e               (retour)

  let x = e; block       (asignement)

  force thunk(x) = e; block   (forçage de thunk, à utiliser avec thunk(e))

  open closure(x) = e; block  (ouverture de fermeture, à utiliser avec closure(e))

  open exp(x) = e; block    (ouverture de fermeture partagée)

decl ::=

  data K(A_1 : ±, ..., A_n : ±) =
    | k_1(T_1+, ..., T_n+)
    | k_2(U_1+, ..., U_n+)
    | ...
  end

  comput K(A_1 : ±, ..., A_n : ±) =
    | m_1(T_1+, ..., T_n+) -> T-
    | m_2(U_1+, ..., U_n+) -> U-
    | ...
  end

  type K(A_1 : ±, ..., A_n : ±) = T

  decl type K : (±, ..., ±) -> ±
```

```
  decl x : T

program ::=

  decl
  decl
  ...
  decl
  block
```