

SORBONNE UNIVERSITÉ

RAPPORT FINAL

19 Mai 2023

Projet STL : Interface pour Autobill

Auteurs :

Brahima DIBASSI
Zeid FAZAZI
Yukai LUO

Encadrants :

Hector SUZANNE
Emmanuel CHAILLOUX



**SORBONNE
UNIVERSITÉ**

Table des matières

| | | |
|----------|---|-----------|
| 1 | Contexte du projet | 2 |
| 1.1 | Historique et définitions | 3 |
| 1.2 | Qu'est-ce qu'Autobill? | 3 |
| 1.3 | Objectifs du projet | 4 |
| 1.4 | Processus de conception | 4 |
| 2 | Interface web | 5 |
| 2.1 | Client | 5 |
| 2.1.1 | Interopérabilité Web-Ocaml | 5 |
| 2.1.2 | L'éditeur de code | 7 |
| 2.1.3 | La résolution de contraintes | 9 |
| 2.2 | Serveur + client | 11 |
| 2.2.1 | Contexte | 11 |
| 2.2.2 | MiniML | 12 |
| 2.2.3 | La résolution des contraintes | 13 |
| 3 | MiniML | 15 |
| 3.1 | Description de MiniML | 15 |
| 3.2 | Caractéristiques de MiniML | 15 |
| 3.3 | Grammaire | 16 |
| 3.4 | Call-By-Push-Value | 19 |
| 3.5 | Sémantique de traduction | 19 |
| 3.5.1 | Schémas de compilation | 19 |
| 3.5.2 | Exemple de traduction | 23 |
| 3.6 | Implémentation | 24 |
| 4 | Conclusion | 25 |
| 4.1 | Avancement du projet | 25 |
| 4.2 | Difficultés rencontrés | 26 |
| 4.3 | Ce qu'on a appris | 26 |
| 4.4 | Pour aller plus loin... | 26 |
| | Références | 28 |
| | Annexe | 29 |

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'inférence : 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.

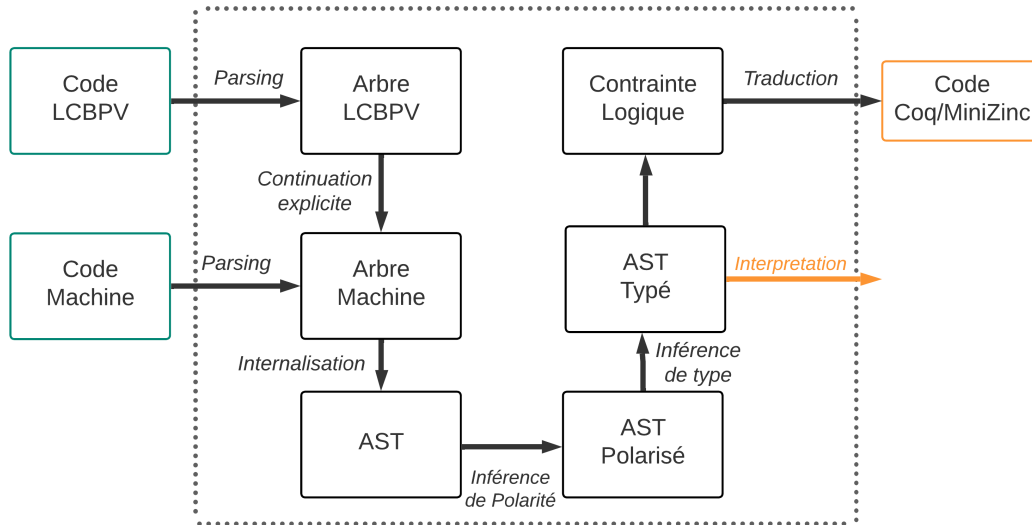


FIGURE 1 – Représentation simplifiée d'Autobill

1.1 Historique et définitions

Des langages de programmation expérimentaux ont vu le jour et ont implémenté cette analyse. On peut citer comme exemple 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 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. Ce mécanisme permet de suivre de manière explicite la 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 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.

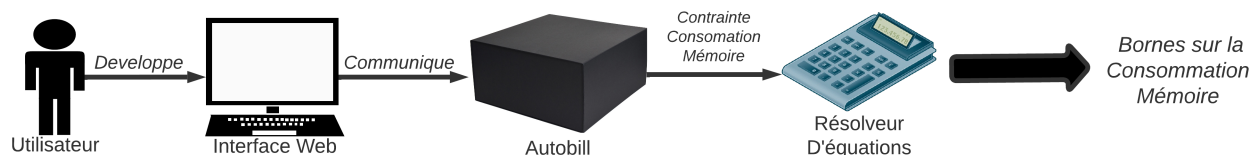


FIGURE 2 – Représentation du système cible

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 chargera également 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 contrainte était qu'il fallait développer cette interface en simultanément 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, on 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 pas 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 800 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 multi paradigme 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 étant de compiler un langage de programmation plus haut niveau, comme C/C++, scheme,.. 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[7] : 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 application 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).

```
1 $ ocamlc -o hello hello.ml
2 $ wasicaml -o hello_wasm hello
```

FIGURE 3 – Chaîne 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` [8]. 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 types 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 (voir Figure 4). 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.

```
1 Js.export
2   "ml"
3   (object%js
4     method translate code =
5       let stderr_buff = Buffer.create 100 in
6       Sys_js.set_channel_flusher stderr (Buffer.add_string
7         stderr_buff);
8       let lexbuf = Lexing.from_string ~with_positions:true (Js.
9         to_string code) in
10      let res = string_of_lcbpv_cst (translate_ML_to_LCBPV
11        lexbuf)
12      object%js
13        val resultat = Js.string res
14        val erreur = Js.string (Buffer.contents stderr_buff)
15      end
16    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[9] ou Rescript[10]. En effet, ce sont tous 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 rencontrer 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[11], 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 avons 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

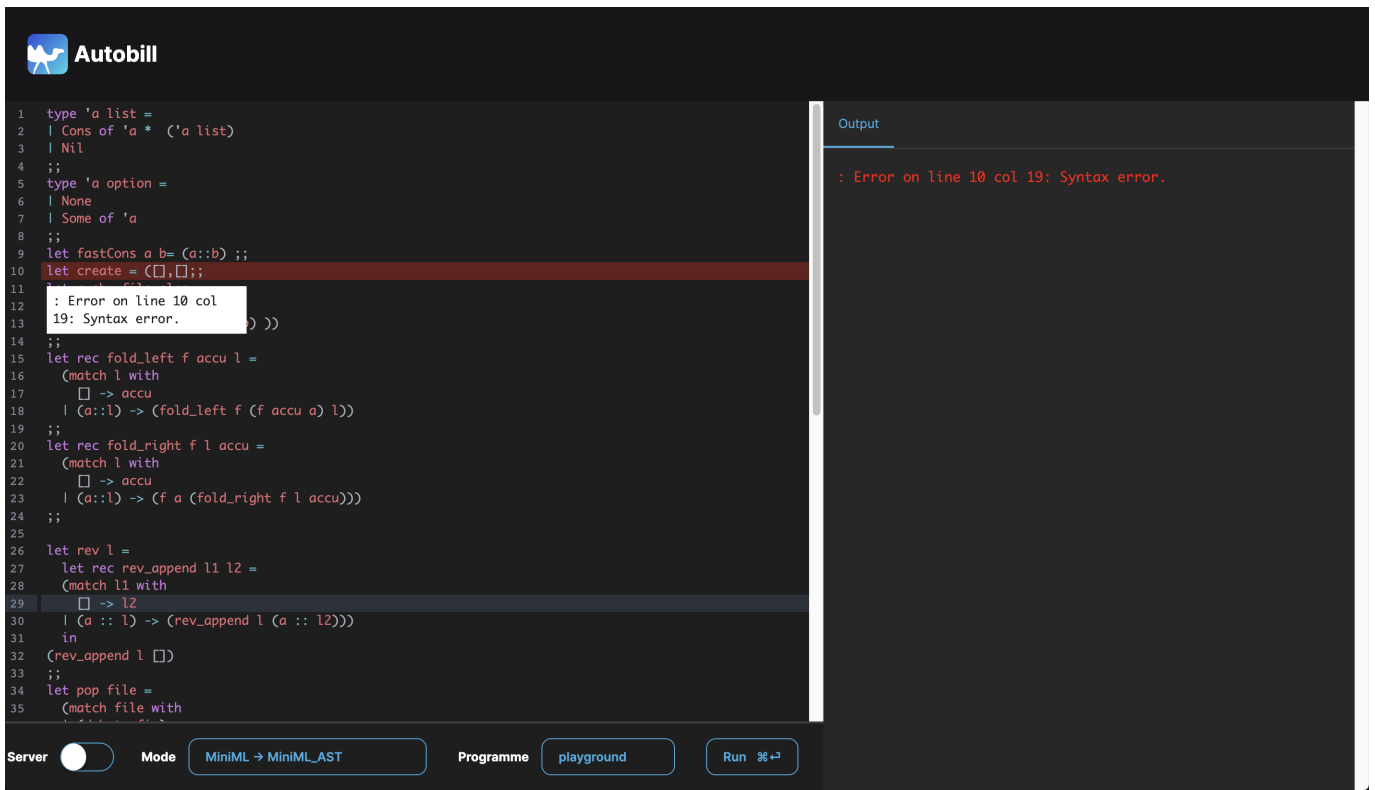


FIGURE 5 – Interface pour Autobill, dans le scénario d’une erreur

afin de les intégrer dans notre projet.

Parmi les bibliothèques disponibles, nous avons choisi CodeMirror[12]. 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 5, 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 6). 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

```

1 {
2   "loc": {
3     "beginning": { "line": 10, "column": 19},
4     "end": {"line": 10, "column": 21}
5   },
6   "info": "Error on line 10 col 19: Syntax error."
7 }

```

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 passé 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 [13] 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.

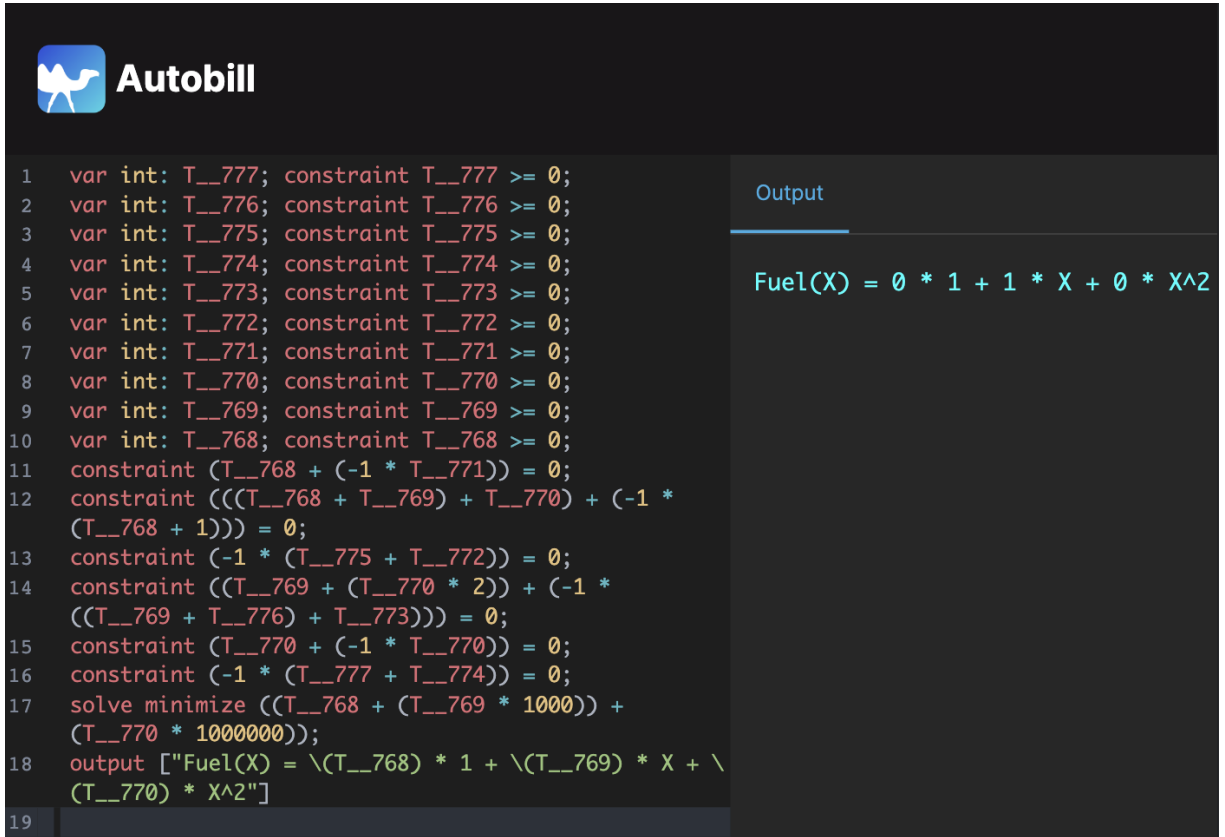
Le support via une librairie Javascript qui utilise elle-même le bytecode WebAssembly est possible. Elle offre une interface entre le développeur et le logiciel MiniZinc via un ensemble d’objets et méthodes qui facilitent les interactions avec l’outil. Comme montré dans la Figure 8, quelques lignes suffisent pour créer son modèle et lancer les tâches de résolution, tout cela nativement dans la machine de l’utilisateur. Cette passerelle permise avec cette API nous permet de faire tourner le moteur de résolution de contraintes directement sur le navigateur et d’afficher les bornes mémoires nécessaires au programme dans l’interface.

```

1      const model = new Model()
2      model.addFile("playground.mzn", code)
3      const solve = await model.solve({
4          options: {
5              solver: "gecode",
6          },
7      })

```

FIGURE 7 – Création et résolution d'un modèle MiniZinc en Javascript



The screenshot shows the Autobill interface with a dark theme. On the left, a MiniZinc model is displayed with line numbers 1 through 19. The model defines several integer variables (T__777 to T__768) with constraints and a minimization objective. On the right, under the 'Output' tab, the resulting polynomial equation is shown: $\text{Fuel}(X) = 0 * 1 + 1 * X + 0 * X^2$.

```

1  var int: T__777; constraint T__777 >= 0;
2  var int: T__776; constraint T__776 >= 0;
3  var int: T__775; constraint T__775 >= 0;
4  var int: T__774; constraint T__774 >= 0;
5  var int: T__773; constraint T__773 >= 0;
6  var int: T__772; constraint T__772 >= 0;
7  var int: T__771; constraint T__771 >= 0;
8  var int: T__770; constraint T__770 >= 0;
9  var int: T__769; constraint T__769 >= 0;
10 var int: T__768; constraint T__768 >= 0;
11 constraint (T__768 + (-1 * T__771)) = 0;
12 constraint (((T__768 + T__769) + T__770) + (-1 *
13 (T__768 + 1))) = 0;
14 constraint (-1 * (T__775 + T__772)) = 0;
15 constraint ((T__769 + (T__770 * 2)) + (-1 *
16 ((T__769 + T__776) + T__773))) = 0;
17 constraint (T__770 + (-1 * T__770)) = 0;
18 constraint (-1 * (T__777 + T__774)) = 0;
19 solve minimize ((T__768 + (T__769 * 1000)) +
(T__770 * 1000000));
output ["Fuel(X) = \"(T__768) * 1 + \"(T__769) * X + \"
(T__770) * X^2"]

```

Output

Fuel(X) = 0 * 1 + 1 * X + 0 * X^2

FIGURE 8 – Affichage de la résolution sur l'interface

Celle-ci prend la forme d'une équation polynomiale qui décrit la complexité spatiale du programme d'entrée. Dans l'exemple de la Figure 8, un unique coefficient non-nul est attaché à x , ce qui suggère une complexité spatiale linéaire pour le programme depuis lequel les contraintes ont été dérivées. On remarque alors la limite dans l'analyse d'Autobill qui ne prend pas encore en compte des complexités logarithmiques ou quasi-linéaires. Des pistes sont encore en cours de développement : il y a notamment le passage des équations sur les contraintes mémoires au format Coq, mais aussi l'ajout de plusieurs raffinements / paramétrages dans la *pipeline* entre MiniML et CBPV (présentée plus bas dans ce rapport) qui permettraient des dérivations de complexités plus précises.

2.2 Serveur + client

2.2.1 Contexte

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, l'utilisation de la librairie core de Mini-Zinc peut échouer en mode client unique. En outre, on ne peut pas garantir que la génération de module Javascript par Js_of_Ocaml soit réussite dans le mode client-seulement. Donc dans l'optique de rendre notre solution plus flexible et *future proof*, une nouvelle version de notre interface, qui déporte les tâches complexes vers un serveur distant, a été développée. Parce que dans une architecture client-serveur, la librairie `child_process` nous propose la solution.

La librairie `child_process` est un module intégré à Node.js qui permet d'exécuter des processus enfants (child processes). Il exploite les fonctionnalités du système d'exploitation sous-jacent pour créer et gérer les processus enfants et utilise des appels système spécifiques pour créer des processus enfants et établir une communication avec eux via des tubes. Il fournit une interface permettant de lancer d'autres programmes ou scripts à partir de notre application Node.js. Cela nous permet d'effectuer des tâches telles que l'exécution de commandes système, l'appel de programmes externes, le démarrage de processus en arrière-plan. Cependant, ce module `child_process` est conçu pour interagir avec le système d'exploitation et exécuter des processus en dehors du navigateur. C'est une fonctionnalité spécifique à Node.js et ne peut pas être utilisé directement dans un environnement de développement front-end, qui s'exécute dans le navigateur. Cette fonctionnalité n'est donc pas disponible pour notre architecture client-seulement. Du coup, nous pensons qu'il soit vital de construire une architecture client-server pour assurer la stabilité et la flexibilité de notre travail.

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.

Comme on a mentionné dans la section 1, l'objectif de notre projet STL est de construire une pipeline du MiniML à la résolution de contrainte (MiniML \rightarrow LCBPV \rightarrow Autobill \rightarrow Équation \rightarrow Résolution). Et nous voulons diviser notre architecture Client-Server en deux parties, la partie MiniML et la partie résolution de contraintes. Les services de la partie MiniML réalisent la partie MiniML \rightarrow LCBPV \rightarrow Autobill \rightarrow Équation de la pipeline, puis les services de la partie résolution de contraintes réalisent la partie Équation \rightarrow Résolution de la pipeline.

Pour cette architecture de client-serveur, on décide d'utiliser la méthode POST de HTTP, pour envoyer une requête qui contient le code écrit dans l'éditeur du code à serveur. Le serveur traite le code soit en utilisant le module généré par Js_of_Ocaml (Pour la partie MiniML, voir section 2.2.2), soit en utilisant la bibliothèque Childprocess, qui peut lancer l'exécution en passant les lignes de commande (Pour la partie Résolution de contraintes, voir section 2.2.3). Après le traitement du code, le serveur renvoie le résultat au client.

2.2.2 MiniML

Dans la partie MiniML, on utilise la méthode POST de HTTP, pour envoyer une requête qui contient le code écrit dans l'éditeur du code à serveur. Le serveur traite le code soit en utilisant le module généré par Js_of_Ocaml. Après le traitement du code, le serveur renvoie le résultat au client. 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 envoie le code du MiniML au serveur via une requête POST. Le serveur le transforme vers l'AST de MiniML et le renvoie au client. Et pour le service *'/MiniML/toLCBPV'*, le client envoie le code du MiniML au serveur via une requête POST. Le serveur le transforme vers l'AST de LCBPV et le renvoie au client. Ces deux services génèrent l'AST de MiniML et de LCBPV. Le côté client affiche l'AST sur le côté après avoir reçu la réponse HTTP. L'objectif principal de ces deux services est de valider les résultats du lexer, du parser et de la traduction, en nous aidant à finir l'écriture des ces trois parties.

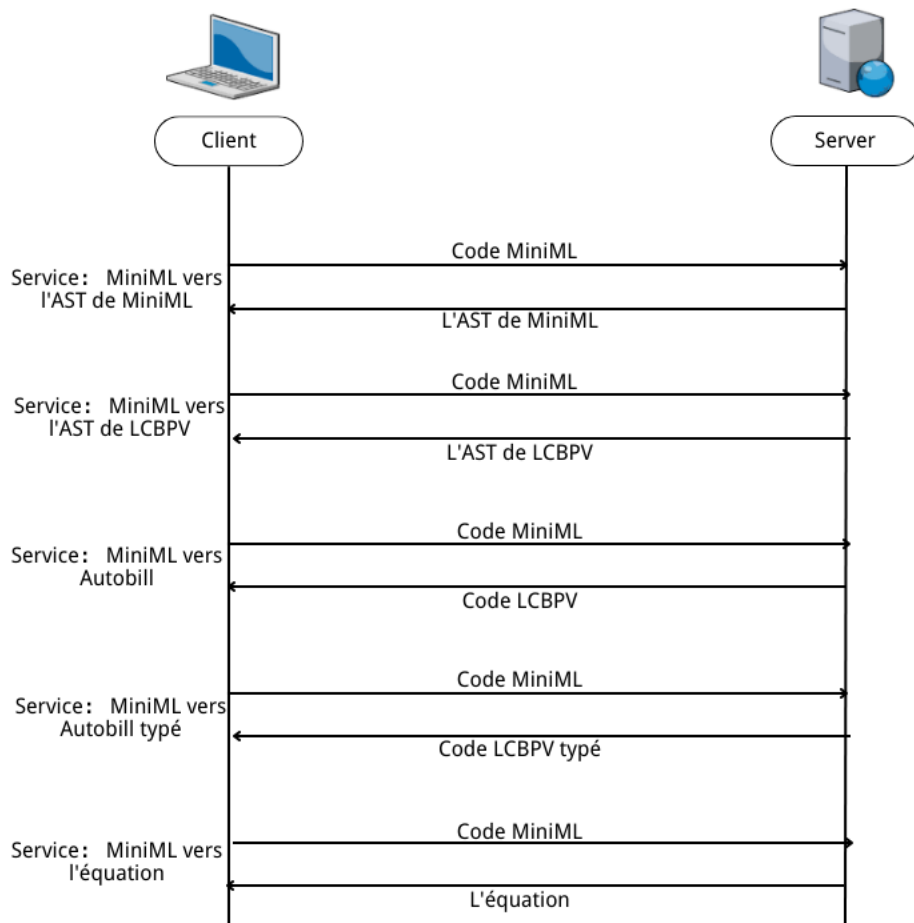


FIGURE 9 – Schéma de communication : MiniML

Pour le service *'/MiniML/toAutobill'*, le client envoie le code du MiniML au serveur via une requête POST. Le serveur le transforme vers sa version internalisé en code machine et le renvoie au client. Et pour le service *'/MiniML/toAutobillType'*, le client envoie le code du MiniML au serveur via une requête POST. Le serveur le transforme vers le code machine d'Autobill typé et le renvoie au client. La côté client affiche le code LCBPV généré par le serveur après reçu la réponse HTTP. Ce deux services génèrent le code de LCBPV traduire par MiniML, cela fournit les sources d'entrée d'Autobill et complète la partie MiniML \rightarrow LCBPV \rightarrow Autobill dans le pipeline.

Pour le service *'/MiniML/toEquation'*, le client envoie le code du MiniML au serveur via une requête POST. Le serveur le transforme vers l'équation résultant de l'analyse statique et le renvoie au client. Ce service permet de générer les équations de l'analyse statique, dans format de code MiniZinc, en utilisant le service correspondant dans Autobill. Ces équations seront utilisées pour obtenir les bornes de consommation de mémoire dans les services suivants, mentionnés dans la section 2.2.3. Ce service complète la partie MiniML \rightarrow LCBPV \rightarrow Autobill \rightarrow Équation dans la pipeline. Mais ce service correspondant dans Autobill est encore en progrès, H.Suzanne est en train de l'améliorer pour obtenir le meilleur résultat. Donc finalement on ne peut pas encore obtenir les équations directement depuis le code MiniML.

2.2.3 La résolution des contraintes

Pour la partie la résolution des contraintes, on utilise la méthode POST de HTTP, pour envoyer une requête qui contient le code écrit dans l'éditeur du code à serveur. Le serveur exécute le code et renvoie le résultat au client. Donc l'objectif de cette partie est l'exécution des équations en format MiniZinc généré par Autobill, et l'obtention des bornes de consommation de mémoire de programme. Pour exécuter le code MiniZinc, on a trouvé deux moyens. Le premier est d'utiliser la librairie de MiniZinc généré par bytecode WebAssembly comme mentionné dans la section 2.1.3. Et la deuxième manière utilise la librairie `child_process`, qui nous permet d'effectuer l'exécution des programmes MiniZinc en passant les lignes de commande.

D'abord on écrit le code dans un fichier temporaire. Et on prépare une ligne de commande qui exécute un fichier MiniZinc en le passant à 'Gecode', un solveur externe de MiniZinc. Puis on utilise la méthode `exec()` qui permet d'exécuter notre commande dans un shell système et de récupérer la sortie dans un tampon. Elle prend en paramètre la commande à exécuter et une fonction de callback. La fonction de callback reçoit éventuellement une erreur, la sortie standard et la sortie d'erreur du processus. (voir Figure 10) Dans la partie résolution de contraintes, nous avons un service *'/minizinc/gecode'* qui exécute le code MiniZinc (Voir Figure 11). Le client envoie le code de MiniZinc au serveur via une requête POST. Le serveur exécute le code, et renvoie le résultat au client.

```

1  const fs = require('fs');
2  const tmpFile = './temp.mzn';
3  fs.writeFileSync(tmpFile, code);
4  const cmd = `minizinc --solver Gecode ${tmpFile}`;
5  exec(cmd, (error, stdout, stderr) => {
6      fs.unlink('temp.mzn', (err) => {
7          if (err) throw err;
8      });
9  })

```

FIGURE 10 – Exécution de code Minizinc

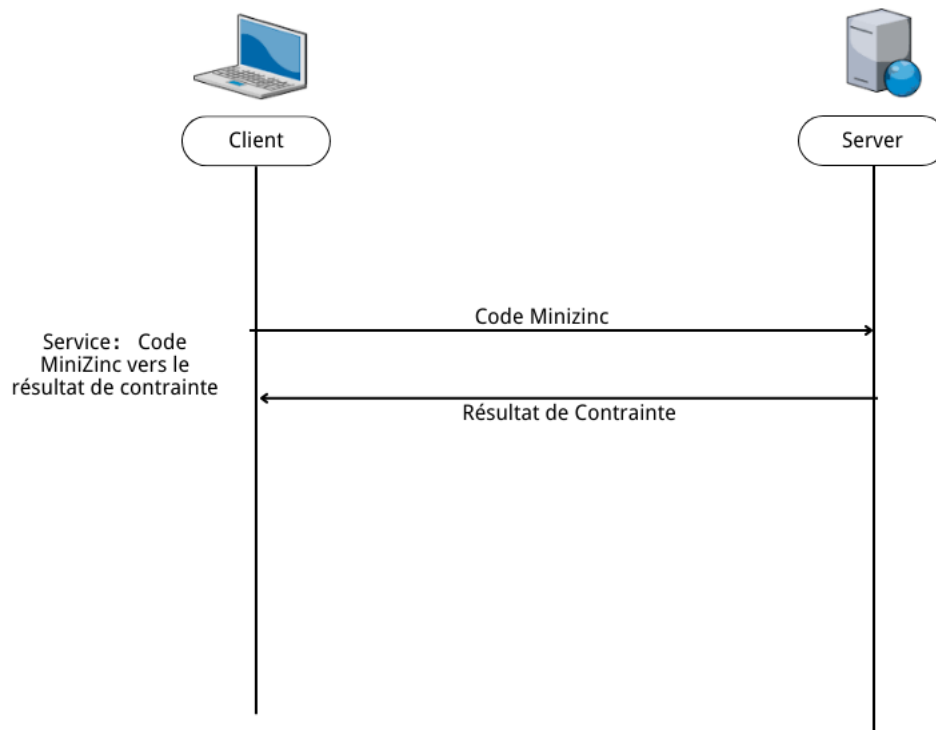


FIGURE 11 – Schéma de communication : MiniZinc

3 MiniML

3.1 Description de MiniML

MiniML émerge du choix de 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 Caractéristiques de MiniML

Une fois les besoins de nos encadrants définis, nous avons pu définir les caractéristiques de **MiniML**. Il s'agit d'un langage fonctionnel pur représentant uniquement la partie fonctionnelle d'OCaml. Pour ce faire, nous avons premièrement mis en place en ensemble minimal permettant un λ -calcul simple et efficace. **MiniML** permet donc comme toute implémentation du λ -calcul simple :

- L'utilisation de variables
- L'application de fonctions
- La définition de fonctions
- La définition de fonctions récursives

Ensuite pour permettre une utilisation plus simple et plus proche de la famille ML, nous avons ajouté la notion de types tel que décrite dans le **Système F** [14][15].

MiniML devient donc un λ -calcul typé de la famille ML grâce à l'ajout en autres des fonctionnalités suivantes :

- La définition de types algébriques avec les types somme et produit et la possibilité de les imbriquer
- La définition de constructeurs de types algébriques
- La définition et l'utilisation de types polymorphes ¹
- Des types de base : **entier**, **booléen**, **singleton** ²

Une fois les caractéristiques de **MiniML** définies, nous avons pu commencer à travailler sur la spécification de ce dernier, en commençant par la grammaire.

1. Par manque de temps dû à quelques essais infructueux, nous n'avons pas pu implémenter les annotations de types rendent peut-être possible la création de problèmes de typage indécidables.

2. On notera l'absence de certains types de base tel que les flottants. Ces derniers n'étant pas pris en charge à l'heure actuelle par **Autobill**.

3.3 Grammaire

Voici la grammaire BNF du langage MiniML. Pour des raisons de lisibilité, tous les éléments de sucre syntaxiques ne seront pas détaillés ici. Les extensions non compatibles avec OCaml mais compatibles avec **Autobill** sont en **Orange**.

Éléments Nommés

On appelle éléments nommés, les identificateurs $\langle \text{Id} \rangle$ pour les variables, motifs et types, les identificateurs de constructeurs $\langle \text{ConstructeurId} \rangle$ et les variables de types $\langle \text{Vartype} \rangle$

$$\begin{aligned}\langle \text{Id} \rangle &::= [\text{'a' - z' 'A' - Z' '0' - 9' '_'}]^+ \\ \langle \text{ConstructeurId} \rangle &::= [\text{'A' - Z' } \langle \text{Id} \rangle \\ \langle \text{Vartype} \rangle &::= \text{'\"'} [\text{'a' - z' } [\text{'0' - 9' }]^* \end{aligned}$$

Programmes

On définit un programme **MiniML** comme une expression précédée de zéro ou plusieurs définitions nécessaires à son évaluation.

$$\langle \text{Prog} \rangle ::= \begin{array}{l} | \langle \text{Expr} \rangle \\ | \langle \text{Def} \rangle \text{';;'} \langle \text{Prog} \rangle \end{array}$$

Définitions

On définit au sein de la règle de grammaire $\langle \text{Def} \rangle$ deux constructions :

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

$$\langle \text{Def} \rangle ::= \begin{array}{l} | \text{'let' } \langle \text{Id} \rangle \text{'=' } \langle \text{Expr} \rangle \\ | \text{'type' } \langle \text{Vartype} \rangle^* \langle \text{Id} \rangle \text{'=' } \langle \text{NewConstructors} \rangle \end{array}$$

Définitions de Constructeurs

On définit au sein de la règle de grammaire $\langle \text{NewConstructors} \rangle$ deux constructions :

- 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 permettant de définir des constructeurs de types algébriques avec des équations.

$$\begin{array}{lcl} \langle \text{NewConstructors} \rangle & ::= & \begin{array}{l} | \langle \text{ConstructeurId} \rangle \text{'of' } \langle \text{Type} \rangle \\ | \langle \text{NewConstructors} \rangle \text{'/' } \langle \text{NewConstructors} \rangle \\ \text{Debut Extension} \quad | \langle \text{ConstructeurId} \rangle \text{'of' } \langle \text{Type} \rangle \text{'with' } \langle \text{ParamAssigns} \rangle \end{array} \end{array}$$

$$\langle \text{ParamAssigns} \rangle ::= \begin{array}{l} | \langle \text{VarType} \rangle \text{'=' } \langle \text{ParmExpr} \rangle \end{array}$$

| | | |
|----------------------|--|-----------------------------------|
| | | <ParamAssigns> ‘,’ <ParamAssigns> |
| <ParmExpr> ::= | | ‘1’ |
| | | <VarType> |
| | | <ParmExpr> ‘+’ <ParmExpr> |
| <i>Fin Extension</i> | | <ParmExpr> ‘*’ <ParmExpr> |

Expressions

On définit au sein de la règle de grammaire <Expr> 10 constructions :

- Les **Littéraux**.
- Les **Utilisations de Variables**.
- Les **Appels d’opérateurs** unaire et binaire.
- Les **Appels de fonctions**.
- Les **Tuples**.
- Les **Lambda**.
- Les **Fonctions Récursives**.
- Les **Constructions**.
- Les **Correspondance de motifs**.

| | | | |
|------------------|-----|--|--|
| <Literal> | ::= | | [‘0 - 9’]+ |
| | | | [‘true’ ‘false’] |
| | | | ‘(‘ ’’ |
| <Expr> | ::= | | <Literal> |
| | | | <Id> |
| | | | <UnaryOperator> <Expr> |
| | | | <Expr> <BinaryOperator> <Expr> |
| | | | <Expr> <Expr> |
| | | | <Expr> ‘,’ <Expr> |
| | | | ‘fun’ <Id> ‘->’ <Expr> |
| | | | ‘fun’ ‘rec’ <Id> <Id> ‘->’ <Expr> |
| | | | <ConstructeurId> <Expr> |
| | | | ‘match’ <Expr> ‘with’ <MatchCase> |
| <UnaryOperator> | ::= | | ‘not’ |
| <BinaryOperator> | ::= | | [‘and’ ‘or’ ‘+’ ‘-’ ‘/’ ‘%’ ‘*’ ‘<’ ‘>’ ‘=’] |

Filtrage et Motifs

On définit au sein de la règle de grammaire <MatchCase> 4 constructions de patterns :

- Les patterns sur **Littéraux**.
- Les patterns sur **Variables**.
- Les patterns sur **Tuple**.
- Les patterns sur **Constructeurs**.

$$\begin{aligned}
\langle \text{MatchCase} \rangle & ::= & | & \langle \text{Pattern} \rangle \text{ '}>' \langle \text{Expr} \rangle \\
& & | & \langle \text{MatchCase} \rangle \text{ '/' } \langle \text{MatchCase} \rangle \\
\langle \text{Pattern} \rangle & ::= & | & \langle \text{Literal} \rangle \\
& & | & \langle \text{Id} \rangle \\
& & | & \langle \text{Pattern} \rangle \text{ ',' } \langle \text{Pattern} \rangle \\
& & | & \langle \text{ConstructeurId} \rangle \langle \text{Pattern} \rangle
\end{aligned}$$

Types

On définit au sein de la règle de grammaire $\langle \text{Type} \rangle$ 5 constructions :

- Les **Types polymorphique**.
- Les **Utilisation de variables de types**.
- Les **Applications de types**.
- Les **Lambda**.
- Les **Tuples**.

$$\begin{aligned}
\langle \text{Type} \rangle & ::= & | & \langle \text{Vartype} \rangle \\
& & | & \langle \text{Id} \rangle \\
& & | & \langle \text{Type} \rangle \langle \text{Type} \rangle \\
& & | & \langle \text{Type} \rangle \text{ '}>' \langle \text{Type} \rangle \\
& & | & \langle \text{Type} \rangle \text{ '*' } \langle \text{Type} \rangle
\end{aligned}$$

Afin de rendre cette règle de grammaire moins abstraite, nous vous fournissons quelques exemples de types valides dans la figure 12.

```

'a list      (* L'application du type list sur 'a *)
'a -> 'b     (* La fonction qui prend un 'a et retourne un 'b *)
int * int    (* Le couple entre deux entiers *)
tuple3      (* L'utilisation du type variable tuple3 *)

```

FIGURE 12 – Exemples de types valide

3.4 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 notions bien distinctes, 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 quand et comment nos calculs vont être effectués.

Dans le cadre de ce projet, trois constructions de LCBPV nous intéressent particulièrement, les **Closures**, les **Thunks** et les **Methods**.

- Les **Thunks** sont des expressions non évaluées, 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**. Annexe : (*Cf. Les thunks*)
- Les **Closures** permettent de capturer un environnement et une expression et de les lier. Annexe : (*Cf. Les fermetures*)
- Les **Methods** qui dans notre cas permettent de définir des calculs à effectuer avec des arguments. Annexe : (*Cf. Les fonctions*)

On remarquera que les **Thunks** et les **Closures** sont deux constructions de valeurs qui contiennent des calculs, la différence fondamentale entre les deux est la suivante :

Les **Thunks** ne prennent pas d'arguments on peut donc les voir comme des calculs prêt à l'exécution. Tandis que les **Closures** sont des calculs qui sont en attente d'arguments.

Toute la subtilité de la sémantique de **Call-By-Push-Value** réside dans la manière dont on va utiliser les trois constructions précédemment décrite pour définir les instants précis où un calcul doit être effectué en séparant sa préparation de son exécution. Pour plus d'informations sur LCBPV, sa syntaxe, sa sémantique et ses constructions, vous pouvez vous référer à l'annexe de ce rapport.

3.5 Sémantique de traduction

Puisque MiniML est un langage *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 équivalence parfaite entre tout comportement de MiniML traduit en LCBPV.

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

3.5.1 Schémas de compilation

Notation

- $\llbracket \text{let } v = e_1 \text{ in } e_2 \rrbracket_{Expr} \rightarrow \text{let } v = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket$
- $\llbracket _ \rrbracket_{Expr} \rightarrow$ est la traduction d'une co expr du langage MiniML vers le langage LCBPV

- À l'intérieur des $\llbracket \cdot \rrbracket$, nous trouvons les éléments propre au langage MiniML qui sont traduit vers le langage LCBPV.
- À l'extérieur des $\llbracket \cdot \rrbracket$, nous trouvons les éléments propre au langage LCBPV.
- On note les sous-constructions de MiniML avec la notation suivante :
 - 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 éléments nommées
 - op pour les opérateurs
 - l pour les littéraux
- $X_1 \dots X_n$ est la liste des sous-constructions de type X allant de 1 à n

Programmes

Pour la traduction des programmes, c'est assez simple, nous allons simplement traduire les définitions et l'expression finale du programme. Et utiliser le mot clef **return** pour retourner le résultat de l'expression finale.

$$\llbracket d_1 \dots d_n e \rrbracket_{Prog} \rightarrow \llbracket d_1 \dots d_n \rrbracket_{return} \llbracket e \rrbracket$$

Définitions

On définit l'opération de traduction $\llbracket \cdot \rrbracket_{Def}$ selon les cas de construction des définitions précisées par la règle de grammaire : **<Def>**

$$\begin{aligned} (\text{VARDEF}) \quad & \llbracket let\ v = e \rrbracket_{Def} \rightarrow let\ v = \llbracket e \rrbracket \\ (\text{TYPDEF}) \quad & \llbracket type\ v_1 \dots v_N = dc_1 \dots dc_N \rrbracket_{Def} \\ & \rightarrow data\ \llbracket v_1 \rrbracket (\llbracket \dots v_N \rrbracket : +) = \llbracket dc_1 \dots dc_N \rrbracket \end{aligned}$$

Lors de la traduction d'une définition de type on remarque l'utilisation de la notation $: +$ pour indiquer que le type défini est un type de valeur. En effet, dans LCBPV il est possible de définir des types de calculs, mais tout type MiniML est un type de valeur pour LCBPV.

Pour plus d'informations sur les types de calculs, vous pouvez vous référer à l'annexe (*Cf. Les autres types de calculs, Définir des types de calculs*) de ce rapport.

Définitions de Constructeurs

On définit l'opération de traduction $\llbracket \cdot \rrbracket_{DefConstructors}$ selon les cas de construction précisées par la règle de grammaire : **<NewConstructors>**

Pour la traduction des définitions de constructeurs, nous allons simplement traduire le type englobé par le constructeur l'opération est la suivante :

$$(\text{MLCONSTRUCTDEF}) \quad \llbracket v\ of\ t \rrbracket_{DefConstructors} \rightarrow v\ \llbracket t \rrbracket$$

On notera l'absence de traduction pour les définitions de constructeurs équationnel car leur support en LCBPV n'est pas encore entièrement spécifiée et implementée. Nous avons donc choisi de ne pas les prendre en compte dans ce rapport et laissons leur traduction en tant que travaux futurs.

Expressions

On définit l'opération de traduction $\llbracket _ \rrbracket_{Expr}$ selon les cas³ de construction des expressions précisées par la règle de grammaire : **<Expr>**

$$\begin{aligned}
(\text{BINARY}) \quad & \llbracket e_1 \text{ op } e_2 \rrbracket_{Expr} \rightarrow \llbracket e_1 \rrbracket \text{ op } \llbracket e_2 \rrbracket \\
(\text{UNARY}) \quad & \llbracket \text{op } e \rrbracket_{Expr} \rightarrow \text{op } \llbracket e \rrbracket \\
(\text{TUPLE}) \quad & \llbracket e_1, e_2 \rrbracket_{Expr} \rightarrow \text{Tuple}(\llbracket e_1 \rrbracket \llbracket e_2 \rrbracket) \\
(\text{LAMBDA}) \quad & \llbracket \text{fun } v \rightarrow e \rrbracket_{Expr} \rightarrow \text{exp}(\text{get} \mid \text{call}(v) \rightarrow \text{thunk}\llbracket e \rrbracket) \\
(\text{FUN REC}) \quad & \llbracket \text{fun rec } v_1 v_2 \rightarrow e \rrbracket_{Expr} \rightarrow \text{exp}(\text{rec } v_1 \text{ is get} \mid \text{call}(v_2) \rightarrow \text{thunk}\llbracket e \rrbracket) \\
(\text{CALL}) \quad & \llbracket e_1 e_2 \rrbracket_{Expr} \rightarrow \{ \\
& \quad \text{open exp } v_1 = \llbracket e_1 \rrbracket \\
& \quad \text{force thunk}(v_2) = (v_1).\text{call}\llbracket e_2 \rrbracket \\
& \quad \text{return } v_2 \\
& \} \\
(\text{CONSTRUCT}) \quad & \llbracket v e \rrbracket_{Expr} \rightarrow v \llbracket e \rrbracket \\
(\text{MATCH}) \quad & \llbracket \text{match } e \text{ with } p_1 \dots p_N \rrbracket_{Expr} \rightarrow \text{match } \llbracket e \rrbracket \text{ with } \llbracket p_1 \rrbracket \dots \llbracket p_N \rrbracket \text{ end}
\end{aligned}$$

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

1. À l'aide de l'instruction **get**, on crée une méthode **call** qui prend en paramètre un argument.
2. Ensuite dans le corps de cette méthode on crée un **Thunk** qui contient la traduction du corps de la fonction.
3. Puis on crée une **Closure** qui contient la méthode 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'évaluation immédiate du corps de la fonction.

3. On ne prend pas en compte les **Littéraux** et les **Variables** dans ce descriptif car leurs traductions sont directes.

Types

On définit l'opération de traduction $\llbracket _ \rrbracket_{Type}$ selon les cas de construction des types précisés par la règle de grammaire : **<Type>**

$$\begin{aligned} (\text{TVAR}) \quad & \llbracket v \rrbracket_{Type} \rightarrow v \\ (\text{TAPP}) \quad & \llbracket t_1 t_2 \rrbracket_{Type} \rightarrow \llbracket t_1 \rrbracket \llbracket t_2 \rrbracket \\ (\text{TCLOS}) \quad & \llbracket t_1 \rightarrow t_2 \rrbracket_{Type} \rightarrow \text{Exp}(\text{Fun} \llbracket t_1 \rrbracket \rightarrow \text{Thunk} \llbracket t_2 \rrbracket) \\ (\text{TTUPLE}) \quad & \llbracket t_1 * t_2 \rrbracket_{Type} \rightarrow \llbracket t_1 \rrbracket \llbracket t_2 \rrbracket \end{aligned}$$

Le cas de traduction le plus intéressant sont ceux des **Types Lambda**. Dans la même idée que les **Expr Lambda**, on dit que la traduction d'un type lambda est un **Type Closure** qui contient un calcul prenant en paramètre un type et renvoyant un **Thunk** sur le type du corps de la fonction.

Motifs et Filtrage

On définit l'opération de traduction $\llbracket _ \rrbracket_{Case}$ selon les cas de construction des expressions précisées par la règle de grammaire : **<MatchCase>**

$$\begin{aligned} (\text{PATTERN LIT}) \quad & \llbracket l \rightarrow e \rrbracket_{MatchCase} \rightarrow \llbracket l \rrbracket \rightarrow \llbracket e \rrbracket \\ (\text{PATTERN VAR}) \quad & \llbracket v \rightarrow e \rrbracket_{MatchCase} \rightarrow v \rightarrow \llbracket e \rrbracket \\ (\text{PATTERN TUPLE}) \quad & \llbracket v_1, v_2 \rightarrow e \rrbracket_{MatchCase} \rightarrow \text{Tuple}(v_1 \ v_2) \rightarrow \llbracket e \rrbracket \\ (\text{PATTERN CONSTR}) \quad & \llbracket v_1 \ v_2 \rightarrow e \rrbracket_{MatchCase} \rightarrow v_1 \ v_2 \rightarrow \llbracket e \rrbracket \end{aligned}$$

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 langage cible.

3.5.2 Exemple de traduction

Voici un exemple partiel de traduction d'un programme **MiniML** vers **LCBPV**. Dans un souci de lisibilité, nous avons ne pas repris la totalité des opérations de traduction seulement les plus intéressantes.

1. On commence par traduire les définitions de types.

```
type type 'a list
```

Deviens :

```
data List (A : +)
```

2. Ensuite nous traduisons chacun des types construits.

```
| Cons of 'a * ('a list)
```

Deviens donc :

```
| Cons(A, List(A))
```

3. On traduit les définitions de variables.

```
let addHead a =  
  (match ls with  
  | [] -> []  
  | (hd :: tail) -> ((hd + a) :: tail))
```

Deviens donc :

```
let addHead =  
  exp(get  
    | call(a) ->  
      thunk(// BODY)  
    end)
```

```
type 'a list =  
| Cons of 'a * ('a list)  
| Nil  
;;
```

```
let ls = [1;2] ;;  
let addHead a =  
  (match ls with  
  | [] -> []  
  | (hd :: tail) -> ((hd + a) :: tail))  
;;  
(addHead 3 ls)
```

FIGURE 13 – Exemple de Code Source MiniML

Comme décrit dans la section 3.5.1, le corps de la fonction est traduit en une **Closure** qui va capturer la liste **ls** et un calcul simulant le corps de la fonction.

4. Une fois nos définitions traduites on peut procéder à la traduction de l'expression finale

(addHead 3 ls)

Deviens donc :

```
return {  
  open exp(mLTempVar2) = {  
    open exp(mLTempVar4) = simpleFunc;  
    force thunk(mLTempVar5) = (mLTempVar4  
    return mLTempVar5  
  };  
  force thunk(mLTempVar3) = (mLTempVar2).call(simpleLs);  
  return mLTempVar3  
};
```

Comme décrit dans la section 3.5.1, l'expression finale qui est une application de fonction est traduite en une ouverture de **Closure** qui va restituer l'environnement original de la fonction et puis une application du calcul sur les arguments de la fonction ayant pour la préparation de ce dernier. Et enfin la restitution du résultat de l'évaluation du calcul.

3.6 Implémentation

Dans le cadre de ce projet, MiniML dispose d'une implémentation écrite en OCaml, langage que nous avons utilisé pour deux raisons principales. D'une part, permettre une meilleure intégration avec Autobill qui est écrit dans le même langage ce qui a permis d'unifier la gestion d'erreurs. D'autre part permettre la double compatibilité avec les deux architectures cibles, cette forte proximité avec le code d'Autobill a rendu possible la traduction non pas de syntaxe concrète à syntaxe concrète comme décrite ci-dessus mais de syntaxe abstraite à syntaxe abstraite. Nous épargnant ainsi une nouvelle étape d'analyse syntaxique. Deux autres divergences avec la spécification est l'utilisation d'une fonction de traduction récursive terminale par souci d'optimisation et l'ajout de nombreux sucres syntaxiques pour faciliter l'écriture de programmes MiniML.

4 Conclusion

4.1 Avancement du projet

L'objectif de notre projet STL était de construire une *pipeline* complète du MiniML à la résolution de contrainte. C'est à dire tout le traitement du code utilisateur en passant par les étapes suivantes :

$$IDE\ MiniML \rightarrow LCBPV \rightarrow Autobill \rightarrow \acute{E}quation \rightarrow R\acute{e}solution.$$

La partie ($LCBPV \rightarrow Autobill \rightarrow \acute{E}quation$) étant couverte par notre encadrant H.Suzanne notre travail était principalement centré sur deux points du logiciel en l'occurrence

$$(IDE\ MiniML \rightarrow LCBPV) \text{ et } (\acute{E}quation \rightarrow R\acute{e}solution).$$

La première partie, comme vous avez pu le lire tout au long de ce rapport, est entièrement spécifiée et implementée à travers la réalisation de l'interface web et du processus de lecture, traduction et analyse du code utilisateur mise en place dans les deux architectures du projet. Ayant désormais la capacité de produire une entrée pour Autobill à travers des moyens simples d'utilisation, compléter la partie ($\acute{E}quation \rightarrow R\acute{e}solution$) et donc, par conséquent, traiter la sortie d'Autobill était notre second objectif. Dans cette optique, nous avons développé une extension de l'interface permettant de visualiser les résultats du traitement de l'équation par un solveur externe afin d'obtenir les bornes sur la consommation de mémoire du programme dans le service "Équation → Solution".

```
data List (A : +) =
| Cons(A, List(A))
| Nil();
let simpleLs = Cons(1, Cons(2, Nil()));
let simpleFunc =
  exp(get
    | call(a) ->
      thunk(match simpleLs with
        | Nil() -> Nil()
        | Cons(hd, tail) -> Cons(hd + a, tail)
      end)
  end);
return {
  open exp(mLTempVar2)= {
    open exp(mLTempVar4)= simpleFunc;
    force thunk(mLTempVar5) = (mLTempVar4).call(3);
    return mLTempVar5
  };
  force thunk(mLTempVar3) = (mLTempVar2).call(simpleLs);
  return mLTempVar3
};
```

FIGURE 14 – Résultat de la traduction

Ce qui complète la partie (*Équation* \rightarrow *Résolution*) de la pipeline elle aussi implémentée dans les deux architectures du projet.

Cependant, la partie (*Autobill* \rightarrow *Équation*) est toujours en cours de développement. En effet H.Suzanne travaille encore sur les détails d'Autobill afin d'obtenir une équation permettant d'augmenter la précision des bornes de mémoire. C'est donc avec regrets que nous ne pouvons finalement pas construire la pipeline entière, pour obtenir la borne de consommation mémoire à partir du programme MiniML. Mais notre avancement de ce projet STL a pu répondre à la plupart des besoins de la section 1 du rapport.

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 [16] pour toute la partie MiniML et le processus de transformation vers CBPV. Puis, les cours de programmation concurrente répartie, réactive et réticulaire [17], notamment pour la partie réticulaire et l'architecture d'applications Web modernes.

4.2 Difficultés rencontrés

Nous avons également rencontré quelques difficultés pour faire avancer le projet. Tout d'abord, nous n'avons pas été en mesure de construire le pipeline complet, avec le lien *Autobill* \rightarrow *Équation* toujours en cours de construction et finalement nous ne pouvons donc pas comparer notre solution avec celle de J.Hoffmann et l'interface de RAML. Deuxièmement, notre projet avançant en parallèle avec H.Suzanne et nous devions être en échange constant avec ce dernier, et nous accommoder mutuellement des progrès réalisés par les deux parties. Une dernière difficulté, et non des moindres, fut la gestion de plusieurs projets en simultané dû à ce semestre de Master 1 qui fut chargé en travail et en projets.

4.3 Ce qu'on a appris

Cependant, ce projet nous a beaucoup apporté. Tout d'abord, notre capacité à développer des projets en partant d'un cahier des charges à la fois précis et en mouvement ressort grandement renforcée, de plus au contact de nos deux encadrants nous avons acquis une bien meilleure compréhension des principes de compilation et de spécification des langages de programmation. Deuxièmement, nous avons appris à composer, analyser et utiliser les connaissances d'un certain nombre d'articles pointus dans lesquels nous avons appris la grande contribution de J.Hoffmann à l'étude de l'analyse automatique des ressources amorties et les principes de call-by-push-value. Ensuite, notre capacité à gérer le temps et à travailler en asynchrone sur plusieurs projets en ressort elle aussi grandement améliorée. Enfin, nous avons appris la grande importance de l'étape de spécification et de documentation dans un projet d'une telle ampleur, et la nécessité de bien comprendre les besoins de l'utilisateur final pour pouvoir lui fournir une solution adaptée à ses besoins.

4.4 Pour aller plus loin...

Le projet est aujourd'hui 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. Dans le futur, quand une pipeline entière pourra être mise en place il sera intéressant

à titre de démonstration de comparer la solution d' H.Suzanne avec celle de Jan Hoffmann et l'interface de RAML [3] avec notre interface Web lorsque celle-ci sera déployée.

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, “Thèse sur call-by-push-value.” <https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf>, 2001.
- [5] G. TACK and P. J. STUCKEY, “Minizinc.” <https://www.minizinc.org/>, 2023.
- [6] INRIA, “The coq proof assistant.” <https://coq.inria.fr/>, 2023.
- [7] REMIXLABS, “Wasicaml.” <https://github.com/remixlabs/wasicaml>.
- [8] OCSIGEN, “Js_of_ocaml.” https://ocsigen.org/js_of_ocaml/latest/manual/overview, 2023.
- [9] J. WALKER, “Reasonml.” <https://reasonml.github.io/>, 2023.
- [10] H. ZHANG, “Rescript.” <https://rescript-lang.org/>, 2023.
- [11] FACEBOOK, “React.” <https://react.dev/>, 2023.
- [12] M. HAVERBEKE, “Codemirror.” <https://codemirror.net/>, 2023.
- [13] A. ZAKAI, “Emscripten.” <https://emscripten.org/>, 2023.
- [14] J.-Y. GIRARD, “The system f of variable types, fifteen years later,” *Theoretical Computer Science*, vol. 45, pp. 159–192, 1986.
- [15] J.-L. KRIVINE, *Lambda-calcul : types et modèles*, vol. 1. Elsevier Masson, 1990.
- [16] P. MANOURY, “Cours aps.” <https://www-apr.lip6.fr/~manoury/Enseignement/2021-22/APS/index.html>, 2021.
- [17] E. CHAILLOUX, “Cours pc2r.” <https://www-apr.lip6.fr/~chailloux/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 représentation intermédiaire de haut niveau, dans lequel 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 fonctionnalités avancées (comme certains concepts 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 définit 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, soustraction, division euclidienne, reste, et négation. On peut tester pour l’égalité et l’ordre. Le type de valeur des entiers est $\text{Int} : +$

```
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 $\text{Bool} : +$

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

On a un `if/then/else` :

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

Les tuples On les note `tuple(e1, ..., en)`, et ils ont le type `Tuple(T1+, ..., Tn+)`. 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, \dots, e_n des tuples par pattern matching :

```
match e_tuple with
| tuple(x1, ..., xn) -> ...
end
```

Les types sommes Les sommes binaires ont le type de valeur $\text{Sum}(T+, U+)$: $+$. Ces deux constructeurs sont $\text{inj}\{1,2\} : T \rightarrow \text{sum}(T, U)$ et $\text{inj}\{2,2\} : U+ \rightarrow \text{Sum}(T+, U+)$. En général, on peut faire des sommes avec n types T_{1+}, \dots, T_{n+} . On note le type de valeurs correspondant $\text{Sum}(T_{1+}, \dots, T_{n+}) : +$ et on a des constructeurs $\text{inj}\{i,n\}$ 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 filtrage par motif sur les listes. Par exemple, si on a une liste d'entiers 1, l'expression ci-dessous renvoie 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 résultat 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 `e` est une expression quelconque 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 différemment 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 calculs 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` correctement (sans segfault).

Par exemple, dans le code suivant, on crée une fermeture sur un calcul qui renvoi un entier. Comme les entiers sont des valeurs, on crée une fermeture autour d'un thunk 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 toujours 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 $\text{closure}(e) = \text{Closure}(T)$
 : + est une valeur de type de valeur $\text{Closure}(T)$

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 “ $f : x \rightarrow 2x+3$ ” 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 $\text{Fun}(T_1+, \dots, T_n+) \rightarrow T-$, avec chaque argument un type de valeur et le résultat un type de calcul. Les fonctions en LCBPV ne sont pas curriifiés. Une fonction de type $\text{Fun}(A+, B+) \rightarrow C-$ doit recevoir ces deux arguments simultanément. Cette fonction, curriifiée, serait de type $\text{Fun}(A+) \rightarrow \text{Fun}(B+) \rightarrow C-$, ou encore $\text{Fun}(A+) \rightarrow \text{Thunk}(\text{Closure}(\text{Fun}(B+) \rightarrow C-))$.

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 côté paresseux. Si on a une paire paresseuse p avec n , éléments, on appelle la méthode $p.\text{proj}\{i, n\}()$ 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 définit une paire paresseuse en spécifiant chacune des méthodes $\text{proj}\{i, n\}$ dans une clause `get ... end` comme pour les fonctions :

```

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 consomme un élément de liste et renvoie un nouvel itérateur, et une seconde qui termine l'itération quand on croise `nil()`. On définit alors un type `Iter(A, B)` qui consomme 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 verra cela plus bas.

Le reste des fonctionnalités

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 écrirait globalement la même chose en LCBPV, mais en séparant 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 fonction 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 exécution du programme. Pour pouvoir interpréter les langages 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 type `Exp(T-)`, comme pour les fermetures avec `closure` et le constructeur de type `Closure`. - On peut accéder le calcul sous-jacent 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.

Tous 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 `: Tuple Sum Choice Thunk Closure Exp`.
- Le constructeur de type `Fun` avec syntaxe spécifique.
- plus les symboles et mots-clés de la syntaxe ci- dessous.

Syntaxe formelle

Notes : - dans `match`, `get`, `data` et `comput`, les cas sont séparés par des `|`, et le Le premier est optionnel. 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` (conditionnelle)

`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                  (assignment)

  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 :  $\pm$ , ..., A_n :  $\pm$ ) =
    | k_1(T_1+, ..., T_n+)
    | k_2(U_1+, ..., U_n+)
    | ...
  end

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

  type K(A_1 :  $\pm$ , ..., A_n :  $\pm$ ) = T

  decl type K : ( $\pm$ , ...,  $\pm$ ) ->  $\pm$ 

```

```
    decl x : T  
program ::=  
    decl  
    decl  
    ...  
    decl  
    block
```