

PSTL : Interface Web Autobill

Fazazi Zeid

Luo Yukai

Brahima Dibassi

23 mars, 2023

Contents

1	Contexte du projet	3
1.1	Historique	3
1.2	Qu'est-ce que Autobill ?	3
1.3	Objectifs du projet	4
1.4	Processus de Conception	5
2	Interface Web	6
2.1	Client uniquement	6
2.1.1	Design du client	6
2.1.2	Outils et Technologies utilisés	7
2.1.3	Tâches réalisées	7
2.2	Serveur + Client	8
2.2.1	Schéma de Communication	8
2.2.2	Outils et Technologies utilisés	8
2.2.3	Tâches réalisées	9
3	MiniML	10
3.1	Pourquoi MiniML ?	10
3.1.1	Call-By-Push-Value	10
3.2	Description Rapide	10
3.2.1	Dépendances	10
3.3	Contenu Actuel	10
3.4	Un exemple de code MiniML	11
4	Tâches à réaliser	12
4.1	MiniML	12
4.2	Serveur	12
4.3	Client	12
4.4	Tests	12

1 Contexte du projet

1.1 Historique

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

L'analyse statique se réfère au domaine en informatique visant à déterminer des métriques, des comportements ou des erreurs dans un code source. Pour un langage et une syntaxe donnée, on peut fixer des sémantiques d'évaluation, du typage ou, dans le cas de notre problématique, autour de la définition de l'occupation en ressources d'un programme.

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

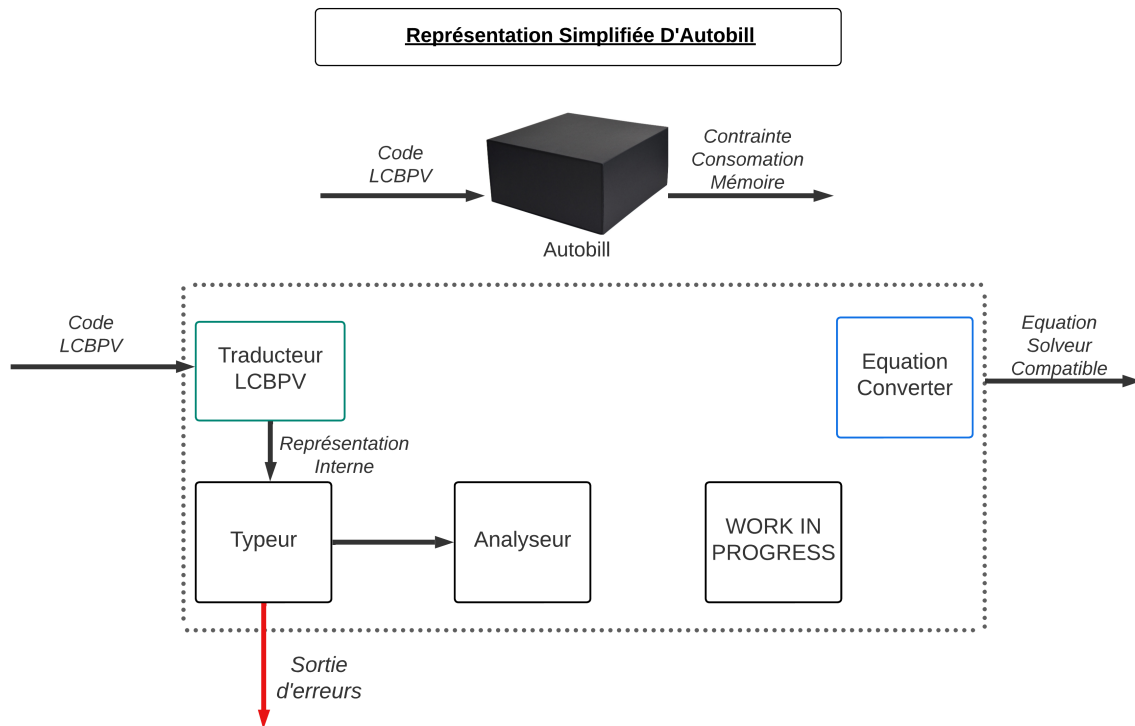
1.2 Qu'est-ce que Autobill ?

La proposition d'Hector avec Autobill se différencie par un niveau d'analyse plus fin. D'abord, Autobill est codé en Ocaml et prend en entrée des programmes écrits soit en modèle machine propre à Autobill, soit en **Call-By-Push-Value**.

C'est un langage qui utilise un paradigme déjà éprouvé, décrit dans les papiers de Paul Blain Lévy [3]. CBPV utilise une pile pour stocker les valeurs et les fonctions manipulées dans le programme. Ainsi, on peut suivre de manière explicite et précise les quantités de mémoire pour chaque valeur allouée / libérée ou fonction appelée / terminée.

Aussi, le langage permet d'exprimer des stratégies d'évaluation dans le code source : on retarde les évaluations jusqu'à ce que les expressions ou les résultats de calculs soient effectivement utilisées et on optimise ainsi la consommation de ressource du programme.

À partir d'une entrée en CBPV, Autobill l'internalise et traduit le programme en un code machine, exprimant explicitement les contraintes matérielles qui s'appliquent sur l'entrée. Enfin, il retourne en sortie ces contraintes formalisées pour satisfaire le format d'entrée de différents solveurs et assistants de preuve, comme MiniZinc ou Coq, afin de prouver des propriétés de complexité temporelle et spatiale.



1.3 Objectifs du projet

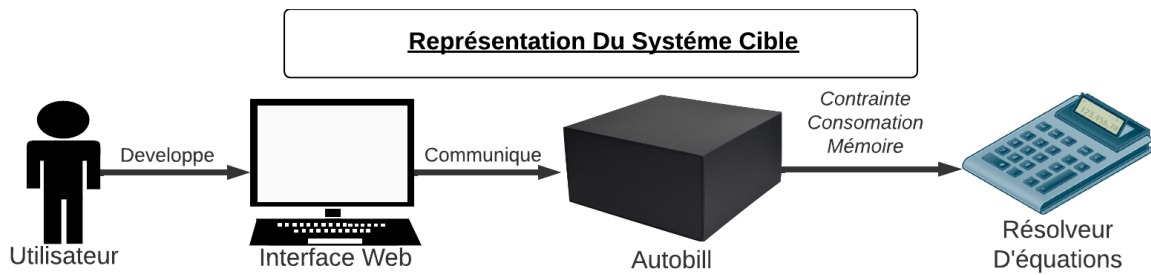
Notre démarche se rapproche de celle faite pour RAML [2] dans leur site officiel.

Le sujet de notre projet STL va donc être de soutenir l'effort de développement en proposant une interface sur le Web permettant la libre manipulation de l'outil Autobill par des tiers 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**. Enfin, on se place aussi sur la sortie d'Autobill en traitant les expressions de contraintes qu'il génère avec des solveurs externes, afin d'en tirer des preuves de complexité et les afficher directement sur le client Web.

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

- L'implémentation du langage MiniML et sa traduction vers LCBPV
- La mise en place d'un client et d'un serveur 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 le langage d'implémentation en effet **Autobill** étant développé en **OCaml**, il était nécessaire de trouver un moyen pour communiquer avec l'application.

La seconde était qu'il fallait développer cette interface en simultané avec **Autobill** et d'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 ses contraintes établies, nous avons du tout au long de ce projet effectuer des choix que ce soit en matière de design ou de technologies.

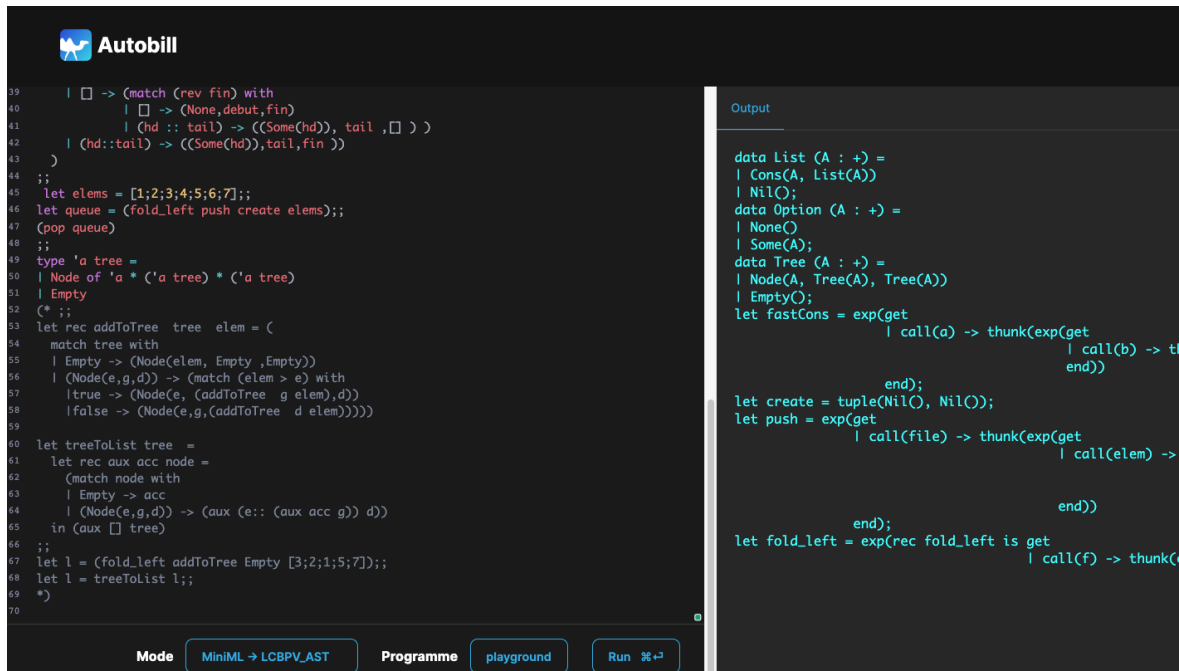
Nous tenons donc à travers ce rapport à mettre en lumière ces décisions tout en décrivant le travail qu'elles ont engendré.

2 Interface Web

Dans l'optique de ne pas se restreindre dans l'utilisation d'outils notamment au niveau du résolveur de contraintes, le groupe s'est orienté vers deux structures de projets différentes et indépendantes : l'une fonctionnant avec un client unique, la seconde avec un serveur dédié et un client qui expose ce serveur. L'avantage réside dans le fait que, lors du développement, si un nouvel outil est amené à être utilisé mais ne dispose de compatibilité sur navigateur Web, alors le serveur peut répondre à ce problème. C'est aussi un sujet de comparaison intéressant à présenter par la suite, que ce soit au niveau des performances que du déploiement de ces solutions.

2.1 Client uniquement

2.1.1 Design du client



The screenshot displays the Autobill web interface, which is a dark-themed code editor. On the left, a file named 'Autobill' is open, showing OCaml code. The code defines a list, a queue, a tree type, and functions for adding elements to the tree and converting it to a list. On the right, the 'Output' panel shows the compiled code, which includes definitions for List, Option, Tree, and functions for fastCons, create, push, fold_left, and treeToList. The interface includes a 'Mode' dropdown set to 'MiniML -> LCBPV_AST', a 'Programme' dropdown set to 'playground', and a 'Run' button with a keyboard shortcut icon.

```
39 | [] -> (match (rev fin) with
40 | [] -> (None,debut,fin)
41 | (hd :: tail) -> ((Some(hd)), tail ,[] ))
42 | (hd::tail) -> ((Some(hd)),tail,fin ))
43 )
44 ;;
45 let elems = [1;2;3;4;5;6;7];;
46 let queue = (fold_left push create elems);;
47 (pop queue)
48 ;;
49 type 'a tree =
50 | Node of 'a * ('a tree) * ('a tree)
51 | Empty
52 (* ;;
53 let rec addToTree tree elem = (
54 match tree with
55 | Empty -> (Node(elem, Empty ,Empty))
56 | (Node(e,g,d)) -> (match (elem > e) with
57 |true -> (Node(e, (addToTree g elem),d))
58 |false -> (Node(e,g,(addToTree d elem))))))
59
60 let treeToList tree =
61 let rec aux acc node =
62 (match node with
63 | Empty -> acc
64 | (Node(e,g,d)) -> (aux (e:: (aux acc g) d))
65 in (aux [] tree)
66 ;;
67 let l = (fold_left addToTree Empty [3;2;1;5;7]);;
68 let l = treeToList l;;
69 *)
70
```

```
data List (A : +) =
| Cons(A, List(A))
| Nil();
data Option (A : +) =
| None()
| Some(A);
data Tree (A : +) =
| Node(A, Tree(A), Tree(A))
| Empty();
let fastCons = exp(get
| call(a) -> thunk(exp(get
| call(b) -> t
end))
end);
let create = tuple(Nil(), Nil());
let push = exp(get
| call(file) -> thunk(exp(get
| call(elem) ->
end))
end);
let fold_left = exp(rec fold_left is get
| call(f) -> thunk(
```

2.1.2 Outils et Technologies utilisés

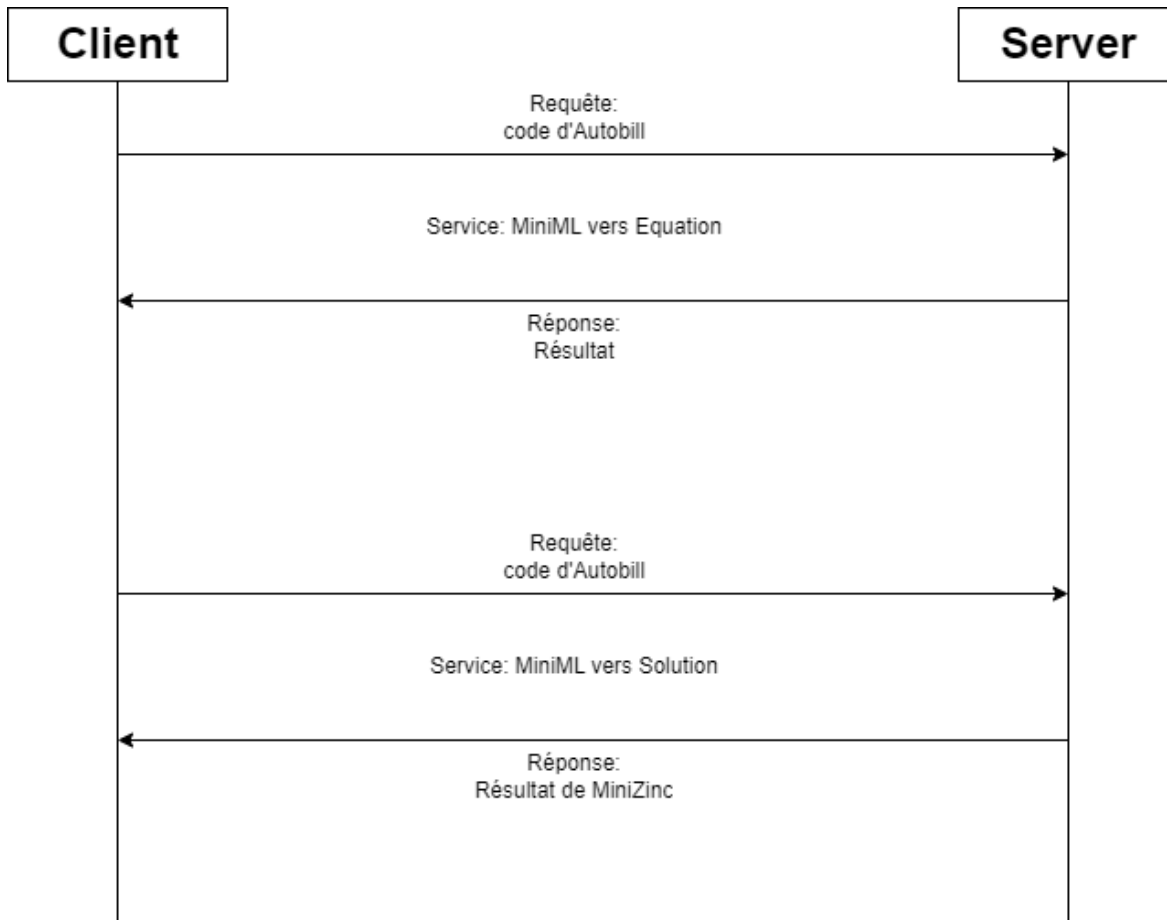
- **HTML / CSS / Javascript** : Il s'agit de la suite de langages principaux permettant de bâtir l'interface Web souhaitée. On a ainsi la main sur la structure de la page à l'aide des balises HTML, du style souhaité pour l'éditeur de code avec le CSS et on vient apporter l'interactivité et les fonctionnalités en les programmant avec Javascript, complété par la librairie React.
- **React.js** : React s'ajoute par-dessus les langages décrits plus haut pour proposer une expérience de programmation orientée composante sur le Web. C'est une librairie Javascript permettant de construire des applications web complexes tournant autour de composants / éléments possédant un état que l'on peut imbriquer entre eux pour former notre interface utilisateur et leur programmer des comportements et des fonctionnalités précises, sans se soucier de la manipulation du DOM de la page Web.
- **CodeMirror** : C'est une librairie Javascript permettant d'intégrer un éditeur de code puissant, incluant le support de la coloration syntaxique, de l'auto-complétion ou encore le surlignage d'erreurs. Les fonctionnalités de l'éditeur sont grandement extensives et permettant même la compatibilité avec un langage de programmation personnalité comme **MiniML**. Enfin, CodeMirror est disponible sous licence MIT.
- **OCaml + Js_of_OCaml** : Afin de manipuler la librairie d'**Autobill**, il est nécessaire de passer par du côté OCaml pour traiter le code en entrée et en sortir des équations à résoudre ou des résultats d'interprétations. Pour faire le pont entre Javascript et OCaml, on utilise Js_of_OCaml, une librairie contenant, entre autres, un compilateur qui transpile du bytecode OCaml en Javascript et propose une grande variété de primitive et de type pour manipuler des éléments Javascript depuis OCaml

2.1.3 Tâches réalisées

- Intégration d'une IDE similaire aux Playground de OCaml et Rescript
- Implémentation d'un éditeur de code supportant la syntaxe de **MiniML**
- Liaison entre le code Javascript et OCaml à l'aide de Js_of_OCaml
- Implémentation de plusieurs modes de traitement du code **MiniML** :
 - Affichage de l'AST MiniML
 - Affichage de l'AST de **Call-By-Push-Value**
 - Affichage de l'Equation résultant de l'analyse statique
 - Vers Representation Interne **Autobill**
- Remontée d'erreurs et affichage dynamique sur l'interface
- Implémentation du solveur d'équations MiniZinc côté client

2.2 Serveur + Client

2.2.1 Schéma de Communication



2.2.2 Outils et Technologies utilisés

2.2.2.1 Coté Client

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

2.2.2.2 Coté Serveur

- **NodeJS**: NodeJS permet une gestion asynchrone des opérations entrantes, ce qui permet d'avoir une grande efficacité et une utilisation optimale des ressources. En outre, NodeJS est également connu pour son excellent support de la gestion des entrées/sorties et du traitement de données en temps réel.

Enfin, la grande quantité de packages disponible sur NPM (le gestionnaire de packages de Node Js) permet de gagner beaucoup de temps de développement et de faciliter notre tâche.

2.2.3 Tâches réalisées

- Intégration d'une IDE similaire aux Playground de OCaml et Rescript
- Implémentation d'un éditeur de code supportant la syntaxe de **MiniML**
- Liaison entre le code Javascript et OCaml à l'aide de Js_of_OCaml
- Implémentation de plusieurs modes de traitement du code **MiniML** :
 - Affichage de l'Equation résultant de l'analyse statique
- Implémentation du solveur d'équations MiniZinc côté client

3 MiniML

3.1 Pourquoi MiniML ?

MiniML émerge de la volonté de créer un langage fonctionnel simple, accessible et sans effets de bord pour les utilisateurs d'autobill car celui-ci requiert une connaissance approfondie de la théorie autour des différentes sémantiques d'évaluation afin de pouvoir manipuler son entrée en **Call-By-Push-Value**.

3.1.1 Call-By-Push-Value

Le paradigme de traitement de langage **Call-By-Push-Value** utilisé par autobill permet à l'aide d'une seule sémantique de traiter deux types de stratégies d'évaluation différentes **Call By Value** utilisée par **OCaml** et **Call By Name** utilisée par **Haskell** pour mettre en place l'évaluation *Lazy*.

Pour permettre cette double compatibilité, **Call-By-Push-Value** effectue une profonde distinction entre les calculs qui font et les valeurs qui sont.

La différenciation entre ses deux types de stratégies s'effectue lors de la traduction depuis le langage d'origine.

3.2 Description Rapide

MiniML dans ce projet dispose d'une implémentation écrite en **OCaml**.

MiniML possède deux types de base (Integer et Boolean).

Il est possible de créer de nouveaux types à partir de ceux-ci.

MiniML est un modeste sous set d'**OCaml** et parfaitement compatible avec un parseur ou compilateur **OCaml**

3.2.1 Dépendances

- **Menhir** : *Menhir* est l'unique dépendance de l'implémentation de **MiniML**, Cette librairie permet la génération de parseurs LR(1) en **OCaml**. *Menhir* est disponible sous une licence GNU GENERAL PUBLIC.

3.3 Contenu Actuel

- Listes
- Fonction Recusives
- Opérateurs de Bases
- Construction de Types
- Variables Globales/Locales
- Files

3.4 Un exemple de code MiniML

```
type 'a option =  
| None  
| Some of 'a  
;;  
  
let createFile = ([],[]);;  
  
let push file elem =  
(match file with  
| (a,b) -> (a,(elem::b)))  
;;  
  
let pop file =  
  (match file with  
  | (debut, fin) ->  
    (match debut with  
    | [] -> (  
        match (rev fin) with  
        | [] -> (None,debut,fin)  
        | (hd :: tail) -> ((Some(hd)), tail ,[])  
      )  
    | (hd::tail) -> ((Some(hd)),tail,fin ))  
  )  
;;  
  
let elems = [1;2;3;4;5;6;7];;  
let queue = (fold_left push createFile elems);;  
(pop queue)
```

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

4 Tâches à réaliser

4.1 MiniML

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

4.2 Serveur

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

4.3 Client

- Retouches esthétiques
- Affichage des erreurs sur plusieurs lignes
- Couverture d'erreurs à traiter la plus grande possible, afin d'éviter les blocages du client
- "Benchmark" la résolution d'équations plus complexes avec le MiniZinc client
- Proposer des programmes d'exemples à lancer, demandant des lourdes allocations mémoires.

4.4 Tests

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

5 Bibliographie

- [1] Hoffmann, Jan, and Steffen Jost. “Two Decades of Automatic Amortized Resource Analysis.” *Mathematical structures in computer science* 32.6 (2022): 729–759
- [2] Levy, Paul Blain. “Call-by-Push-Value: A Subsuming Paradigm.” *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. 228–243
- [3] Hoffman, Jan. Resource Aware ML, Web, URL
- Will Kurt. 2018. *Get Programming with Haskell*. Manning Publications.
- Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002
- Winskel, Glynn. *The Formal Semantics of Programming Languages: an Introduction*. Cambridge (Mass.) London: MIT Press, 1993
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers : Principles, Techniques, and Tools*. 2nd ed. Boston (Mass.) San Francisco (Calif.) New York [etc: Pearson Addison Wesley, 2007]
- Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml*. O’Reilly Media, URL
- Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. *Proceedings of the ACM on Programming Languages* 1, Volume 1, Issue ICFP, Article No.: 43, pp 1–29 (2017)
- Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, Volume 3, Issue ICFP, Article No.: 110, pp 1–30 (2019)
- Xavier Leroy. 2022 OCaml library. OCaml Lazy Doc. Retrieved February 20, 2023 **Link**
- Haskell - Wikibooks, open books for an open world. Doc Haskell. Retrieved February 17, 2023 **Link**
- Emmanuel Chailloux, Pascal Manoury, Bruno Pagano. *Développement d’applications avec Objective Caml*. Paris Cambridge [etc: O’Reilly, 2000], URL
- Hector Suzanne. *Autobill*, 2023, GitLab, URL